A Compiler for Nondeterministic Term Rewriting Systems

Marian Vittek

INRIA-Lorraine & CRIN
615, rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy Cedex, FRANCE

ILOG
9, rue de Verdun, BP 85
94253 Gentilly Cedex, FRANCE

E-mail: vittek@loria.fr, vittek@ilog.fr

Abstract

This work presents the design and the implementation of a compiler for the ELAN specification language. The language is based on rewriting logic and permits, in particular, the combination of computations where deterministic evaluations are mixed with a nondeterministic search for solutions. The implementation combines compilation methods of term rewriting systems, functional and logic programming languages, and some new original techniques producing, in summary, an efficient code in an imperative language. The efficiency of the compiler is demonstrated on experimental results.

1 Introduction

The use of term rewriting in software development is natural in many areas such as computer algebra, constraint solving or compiler development. Several programming languages such as OBJ or ASF+SDF use term rewriting as the engine for the execution of their programs. In these languages the rewriting rules are seen as oriented equations between terms, rewriting itself is a *simplification* of a term inside its equivalence class into the *canonical form*. This point of view is too restrictive for many practical applications. Some more free interpretations of term rewriting rules can be found for example in [JK91, Frü95]. In particular, rewriting logic [MOM93] proposes a semantic background where term rewriting is seen as a sequence of *transitions* which starts with an initial term t and transforms it to its *result form*. In rewriting logic a query of the form $t \rightarrow^* x$ leads to the search of all terms x, such that t can be rewritten to x. Rewriting becomes nondeterministic in the sense that there can be several result forms starting from one initial term.

The use of rewriting logic raises new interesting problems concerning the efficient implementation of *nondeterministic rewriting*. We call nondeterministic rewriting the process that starts from an initial term and looks for all terms to which it can be rewritten. This search is obviously implemented by backtracking where some branches of the search space may be pruned by additional conditions. The implementation of nondeterministic rewriting recalls the implementation of logic programming languages, the significant difference is due to the fact that the rewriting rules can be applied inside the terms. Also the formalism we use to prune the search space is different from that of logic programming languages.

In this paper we discuss the possibilities of efficient implementation of nondeterministic rewriting by compilation into an imperative programming language. In particular, we want to present the techniques we have used in the compiler of the ELAN language [Vit94, KKV95]. ELAN is a logical framework whose specification language is entirely based on term rewriting. It is used for the specification and study of different constraint solvers, logic programming languages, and their combinations. Semantically, it is based on a specialization of rewriting logic called *computational system* [KKV95]. It introduces a notion of *strategy* which permits, in particular, the formalization of algorithms combining deterministic and nondeterministic computations. We will briefly present the language in the following section.

The main originality of the paper is the presentation of the ELAN compiler, namely the first presentation (to the author's best knowledge) of a compilation scheme for nondeterministic rewriting. The presented compiler transforms rewriting systems into C programs compiled then by a standard compiler. A new technique is defined for compiling the specific control flow in programs during the backtracking. Our method preserves the efficiency of deterministic computations and can be of more general interest, for example, it can be interesting to test it in implementations of constraint solvers, of the WAM and Prolog-like languages. The experimental results, presented at the end of the paper, show that the nondeterministic rewriting can be implemented as efficiently as the best current implementations of functional and logic programming languages. Briefly, on classical benchmarks taken from functional programming we have obtained on average results slightly better than SML compiler of New Jersey [AM87]. On other benchmarks, testing backtracking, we have obtained significant speed-up compared to ECLiPSe [MS$^+$93].

The paper is organized as follows: We start by a brief presentation of the ELAN language, then we discuss methods for the efficient compilation of both deterministic and nondeterministic rewriting, we present in particular several optimizations of the generated code and the implementation of the library realizing backtracking. We finish the paper by presentation of some experimental results and by a brief conclusion.

2 The ELAN Language

We present only briefly, on simple examples, the language main features, more detailed description can be found in [Vit94, KKV95]. Similarly to algebraic specification languages like OBJ or ASF+SDF, a *program* is composed from *modules*, each of them defines its own signature, labelled rewriting rules and *strategies* controlling the applications of rules. In the rest of the paper, we use the term "strategy" meaning a piece of code of ELAN's program. We are aware that this term becomes overloaded in the context of the term rewriting, where it has its specific meaning. Terms in ELAN can be written in a mix-fix syntax defined by a many sorted signature. A module can *import* other modules and a *parametrization* is possible and realized (for the moment) by a textual pre-processing. The rewriting rules can be conditional and a special kind of *local assignment* is permitted inside the rules. A special formalism is used to define strategies. Strategies describe sequences of elementary transformations (i.e. applications of rewriting rules) from initial (*input*) terms to resulting (*output*) terms. The strategies are constructed using a particular formalism permitting applying a rule (and testing its applicability), making a loop and sequencing. The rewriting rules from a strategy are applied at the head position of the term which is not really a limitation since applications inside terms can be simulated using local assignments. A program can contain definitions of several strategies, one of them has to be designed as being the *main* strategy. When executing the program, the main strategy is applied on the input term.

The language contains built-in boolean, integer and floating point arithmetics. Based on practical experience, rewrite rules are divided into two groups: the labelled rewrite rules, whose application is controlled by user-defined strategies, and the non labelled rules, whose application is controlled by a default *normalization strategy*, which consists in applying non-labelled rules at leftmost-innermost positions of a term until the normal form is reached. The normalization strategy is used with rules defining deterministic functions and is applied after each reduction produced by a labelled rewrite rule. This corresponds to the intuition that, after each application of a labelled rewrite rule, functions introduced by the right-hand side of the rule must be evaluated.

In the rest of the section we introduce and illustrate on examples some important features of the language.

Examples

Let us take for the first example the *append* function on lists. Let us suppose that we have defined a signature containing the constant *nil* and the binary symbols *cons* and *append*. The following code defines the function appending two lists:

```
[]    append(nil,y)          => y                       end
[]    append(cons(x,y),z)    => cons(x,append(y,z))     end
```

In this syntax, the rewriting rules begin with their label inside `[]` brackets. If there is no label mentioned (like here) the rules are applied according to the built-in normalization strategy. This means that the intended semantics of the evaluation is functional. The key word `end` is used to end definitions of rewriting rules.

In the second example, we define a nondeterministic strategy range which returns the n results: $n, n-1, \cdots, 1$ when applied to an integer n. We define first the following conditional rewriting rule:

```
[range_rule]  x     => x-1      if x>1                 end
```

We use this rule to define the strategy range which, when applied to n, returns n itself, then it returns the result of one application of the rule range_rule on n, then the result of two applications of the rule, etc until the rule is no longer applicable. In this strategy we will use two elementary strategy constructors: **dont care choose** provides an application of the first applicable rewriting rule from a given list and **iterate-enditerate** (a nondeterministic loop constructor) which when applied to a term t returns t itself; then after a backtrack it returns the result of application of its sub-strategy to t; after another backtrack it returns the result of two applications of its sub-strategy, etc, until no more application is possible. In other words, the iteration constructor provides a loop making at each pass a nondeterministic branching between breaking and continuing the loop. The definition of the strategy range is:

```
strategy range
  iterate     dont care choose(range_rule)    enditerate
end of strategy
```

Definitions of strategies begin with the key word `strategy` following by the name of the strategy. Strategies are ended by the phrase `end of strategy`.

A last example consists in a program providing each possible insertions of an element into a given list. For example, an insertion of the term c into the list $a.b.nil$ (the . here is an infix notation for the *cons* operator) gives three results: $c.a.b.nil$, $a.c.b.nil$ and $a.b.c.nil$. In this example we suppose we have defined the binary symbol *insert* and the ternary symbol *ins*, and consider the rewriting rules (where the symbols `e,e1,e2,l,l1,l2` stand for variables):

```
[i1] insert(e , l)          => ins(e , nil , l)                end
[i2] ins(e , l1 , l2)       => append(l1 , e.l2)               end
[i3] ins(e1 ,l1 , e2.l2)    => ins(e1 , append(l1 , e2.nil) ,l2)  end
```

For a given list L and a given element a, all possible insertions of a into L can be obtained by applying first the rule `i1` to the term $insert(a, L)$ and then, while it is possible, and in a nondeterministic manner the rules `i2,i3`. During the loop the transformed term is of the form $ins(e, l1, l2)$ and the invariant of the loop is that $append(l1, l2)$ is equal to the original list L. In other words, successive applications of the third rule cut the original list L into two pieces $l1$ and $l2$. An application of the second rule creates the list $l1.a.l2$ and terminates the loop (as no other rule will be further applicable). The strategy starting with an application of rule `i1` and continuing by repeatedly applying rules `i2,i3` in a nondeterministic manner can be written in ELAN as:

```
strategy insert
  dont care choose(i1)
  while    dont know choose(i2 i3)    endwhile
end of strategy
```

This strategy uses the **dont know choose** construction which provides a nondeterministic application of rewriting rules (exploring all the possibilities) and the **while-endwhile** loop, which applies its sub-strategy until no more application is possible. As opposed to the **iterate** loop, **while** returns only "final" results, it returns only terms to which no further application of its sub-strategy is possible. The strategy `insert` illustrates also the *sequencing* of two sub-strategies, the final results are obtained by application of the **while-endwhile** loop to the results of the **dont care choose** sub-strategy.

Note that other strategies can give the same results as the previous one. One such strategy is for example:

```
strategy insert2
  dont care choose(i1)
  iterate   dont care choose(i3) enditerate
  dont care choose(i2)
end of strategy
```

This strategy first applies rule `i1`, then it iteratively applies rule `i3`, and after each iteration it gives a result by final application of rule `i2`.

3 Compiling Term Rewriting with Strategies

Compilers of term rewriting systems provide a significant speed-up of execution compared to interpreters. In our case, the additional formalism of strategies contain possibilities of nondeterministic computations which makes it more difficult to implement than simple conditional term rewriting system. The design of the general architecture of our compiler is influenced mainly by the search of trade-offs between efficiency of deterministic and of nondeterministic computations. In practical experiences we have observed that even the programs using massive nondeterminism spent a large part of run time in deterministic evaluations. For example, algorithms used in a variety of constraint solvers, provide deterministic propagation of changes after each nondeterministic step. In our compiler we give then priority to the efficient implementation of deterministic computations.

The general idea of our compiler is to transform a given ELAN program into a C program, which receives an input term t and computes the results of application of the program's main strategy to t.

The compiler transforms each strategy S into a C function (named str_S) of one argument. An application of a strategy S to a term t is compiled into a call to the function str_S whose argument is a pointer to t. The function returns the pointer to the resulting term. It is obvious that this scheme of compilation works well for deterministic strategies, in the nondeterministic case, when the strategy S can return several results, the situation is more complicated.

3.1 Deterministic Computations

We present first in this section the method (refined by several optimizations) we use for compilation of deterministic strategies and we discuss the general non-deterministic case later in section 3.2. As we have just mentioned, each strategy S is compiled into a C function *str_S*. This function is built according to the structure of the original strategy, in particular, the sequencing of sub-strategies is compiled into a concatenation of parts of C code and the **while** construction is compiled into a loop. The most interesting thing is the compilation of the elementary strategy **dont care choose**$(\ell_1, \cdots, \ell_n)$. This strategy, when applied to a term t, selects the first of the rules $\ell_1, \cdots, \ell_n$ which can be applied to t, it matches t with the left hand side of the rule, instantiates its variables and replaces t by the right hand side of the selected rule. The naive compilation of this process gives us a sequence of n parts of code corresponding to application of each rewriting rule. Each part starts with a compilation of the matching algorithm for the corresponding rule. A successful matching instantiates the variables of the left hand side and the code continues by evaluating the conditions and the local affectations. Then it continues by freeing the part of the input term corresponding to the left hand side of the rewriting rule and finally by the construction of the right hand side of the rule. If the matching fails, a condition was evaluated to false or there is no local affectation of variables the computation continues on the part corresponding to the next rewriting rule. If no rule is applicable a call of a special function *fail* is generated (see also the section 3.2 of the paper).

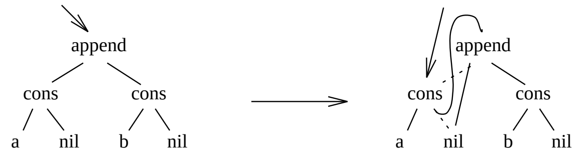
Several refinings of this general schema are possible. We present now those we have implemented in our compiler.

Many to one matching

The first possible optimization concerns the compilation of the matching algorithm. All the rewriting rules are known at compile time. Matching of left hand sides of rules to one unknown input term can be pre-compiled using some existing algorithm. The resulting code gives the set of applicable rules in a very efficient manner. Several works are concerned with this or similar problems, among others [HO82, Aug85, Vit90, KL91]. In the ELAN compiler we have implemented two of them, namely [Aug85] and [Vit90].

Reusing parts of terms

Another optimization comes from the Berlin project OPAL [DFG$^+$94]. The main idea is to reuse parts of left hand sides of rules to construct the right hand sides when applying a rewriting rule. In the case when both sides of a rewriting rule are of approximately the same size, a significant speed-up can be gained. Let us take for example the rewriting rule: $append(cons(x, y), z) \rightarrow cons(x, append(y, z))$ and apply it to the term: $append(cons(a, nil), cons(b, nil))$. In the rewriting rule, the size of the left hand side is exactly the same as the size of the right hand side. Once the rule matches a term, we know that the term to reduce is of the form $append(cons(x, y), z)$. Note that after the reduction this part of the reduced term is of no further use, it is considered as garbage and is freed. We can construct efficiently the resulting term $cons(x, append(y, z))$ by re-using this garbage. The next figure shows that in this case only two affectations of pointers are necessary to provide the reduction.



However, this optimization is not possible if some parts of the reduced term are shared. To detect easily whether a node is shared or not we use reference counters indicating how many times a term is referenced. The reference counters are also used in the garbage collection. The rewriting rules are known during compilation, so the possibilities of reusing their parts can be pre-computed at compile time.

Collecting rules with the same head symbol

A special optimization concerns the non labelled rules which are applied with the built-in normalization strategy. This optimization used for example in [Kap87] consists of regrouping the non labelled rules with the same head symbol on the left hand sides. For each functional symbol f of arity n the collected rules are compiled into a C function fun_f of n arguments. This function when applied to the terms $t_1, \cdots, t_n$ returns the result of applying the normalization strategy to the term $f(t_1, \cdots, t_n)$. Remember that the built-in normalization strategy is used after each application of a rewriting rule. There is no reason to construct first the right hand side of the applied rule and to normalise the resulting term only after this. Using the functions fun_f, both steps can be made together, so the construction of the right hand side can be compiled into nested calls of fun_f functions. This optimization is illustrated on the example in the section 3.1.

Further optimizations

Several further optimizations are implemented in the ELAN compiler. In particular the strong typing of the language permits us to directly represent the numeric values. No additional information is stored in terms to determine whether a value is a pointer or numeric value. Numerical expressions are built-in and they are compiled directly into C expressions. To decrease the time spent to maintain the reference counters, no reference counting is provided on the constants and numeric values. All the functional constants are stored in a special part of the memory, they are never freed and multiple occurences of the same constant are always shared, so at run-time no constant needs to be created. This optimization helps, because in the average case, the constants represent approximately one half of the symbols in a term. The compiler provides also the tail recursion and the tail recursion modulo constructor application optimizations, which are compiled into loops.

Example of the generated code

Let's take two rules for computing the n-th Fibonacci number.

```
  [] fib(x,y,0)         => y                          end
  [] fib(x,y,z)         => fib(y,x+y,z-1)             end
```

These rules have no label, they are applied with the built-in normalization strategy. Using these two rules, any term $fib(0,1,n)$ where n is a natural number will be rewritten to the n-th Fibonacci number. Note also that the normalization strategy always applies the first applicable rule.

Several optimizations come in while compiling these two rules. As the rules are applied with the built-in normalization strategy and they have the same functional symbol fib at the head of left hand sides, they are compiled into a function fun_fib of three arguments. The body of the function starts with the many-to-one matching (in this case only a test whether the third argument is zero or not). Two possibilities correspond to the application of each of these rewriting rules. No reference counting occurs in this example, because all concerned terms are numerical values. For the same reason, no freeing of left hand sides of rules is needed. The tail recursion optimization occurs in the construction of the right hand side of the second rule.

The resulting code in C, and then compiled to SPARC assembler is:

```
#define PLUS(a,b) ((int) a)+((int) b)
#define MINUS(a,b) ((int) a)-((int) b)

struct term *fun_fib(v1,v2,v3)
struct term *v1,*v2,*v3;
{ beginlabel:
  switch((int) v3){
  case 0 :  return(v2);
  default: {
    struct term *sv2,*sv1;
    sv1 = MINUS(v3,1);
    sv2 = PLUS(v1,v2);
    v1 = v2;  v2 = sv2; v3 = sv1;
    goto beginlabel;
} } }
```

```
_fun_fib:
      b      LY2
      tst    %o2
LY1:  mov    %o1,%o0
      mov    %o3,%o1
      dec    %o2
      tst    %o2
LY2:  bne,a LY1
      add    %o0,%o1,%o3
      retl
      add    %g0,%o1,%o0
```

Note, that the normalization using the two rewriting rules is compiled in a cycle where at each pass (corresponding to one application of a rewriting rule) only two register movings, one addition instruction, one decrementation and one test of zero are executed.

3.2 Nondeterministic Search

The nondeterminism in ELAN programs occurs when using one of the strategy constructors: **dont know choose** or **iterate**. It is implemented by backtracking which at the moment, where two rewriting rules can be applied in a nondeterministic manner, applies the first one and continues the execution until a result term is obtained or until some sub-strategy fails. After this, it returns back, it recovers the state of the evaluation where it was at the moment of the choice and it continues the evaluation exploring the second branch.

To compile this specific control flow we have implemented a pair of functions, named setChoicePoint and fail. The function setChoicePoint when called returns zero and the computation continues. Then when the function fail is called, it provides a jump into the last call of setChoicePoint and returns the integer 1. These functions may remind the reader of the pair of standard C functions setjmp and longjmp providing so-called long jump between functions. However, the longjmp can be used only in a function called from the function setting setjmp. Our functions have not such a limitation which makes them more difficult to implement. We present this point in section 3.2 of the paper. In the rest of the paper we talk about *setting a choice point* when calling the function setChoicePoint.

Once having these two functions, the general scheme of compilation of nondeterministic computations becomes easy. Each time we have two possibilities to explore in a nondeterministic manner (for example to apply two rewriting rules ℓ_1, ℓ_2), the compiler generates a call to the function setChoicePoint and a test of its result value. The first possibility (the rule ℓ_1) is then compiled into the branch corresponding to the result zero, the second one (the rule ℓ_2) into the other branch. I.e. the nondeterministic application of two rewriting rules ℓ_1, ℓ_2, gives the code: `if (setChoicePoint()) {` $apply(\ell_2)$ `} else {` $apply(\ell_1)$ `}`.

Nondeterministic strategies

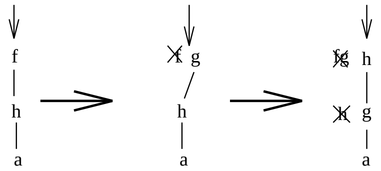
We can now extend the compilation schema from the section 3 and to define how the nondeterministic strategy constructors are compiled. Similarly to its deterministic equivalent, the strategy **dont know choose**$(\ell_1, \cdots, \ell_n)$ is compiled into a sequence of parts, each part corresponding to a compilation of the corresponding rewriting rule. The main difference from the deterministic case is that each part starts with a call to the function setChoicePoint and provides a jump to the next part, if this call returns one. After all parts a call to the function fail is generated. We can remark at this point that the many-to-one matching optimization is possible in the nondeterministic case and is implemented in the current version of the compiler. The **iterate** constructor is compiled into an infinite loop containing the line `if (! setChoicePoint()) break;` followed by the code corresponding to the sub-strategy of **iterate**. The first generated line means that the function setChoicePoint is called inside the loop and the loop is broken if the result of this call is not equal to zero. It is easy to see that the compilation of both strategies satisfies the expected control flow as presented in the section 2.

Saving the working memory

A particular problem arises during the nondeterministic computations. Suppose, for example, that we have defined the rewriting rules:

$$\begin{array}{llcl} \ell_1: & f(x) & \to & g(x) \\ \ell_2: & f(x) & \to & x \\ \ell_3: & g(h(x)) & \to & h(g(x)) \end{array}$$

and we apply the strategy **dont know choose**(ℓ_1, ℓ_2)**dont care choose**(ℓ_3) on the term $f(h(a))$. This evaluation sets the choice point between the first two rules, applies the first one and then the third one. This gives the first result $h(g(a))$. This computation is shown in the following figure:



After having obtained the first result the function fail is executed to explore other possibilities. The execution of fail provokes a jump to the last choice point and the execution has to continue by an application of the second rule ℓ_2 on the original term $f(h(a))$. However, this term does not exist anymore in the memory. The problem is that the original method of compilation is not adapted to nondeterministic computations and it destroys the terms which could be needed later in other branches of the computation.

In our implementation we have solved this problem by a special compilation of the rewriting rules applied by the **dont know choose** strategy. Contrary to the deterministic case such an application do not free the term corresponding to the left hand side of the rewriting rule. This implies on the other hand, that this term would be never freed. For this reason we have introduced a special data structure called *Trail*. The Trail contains pointers to the term on which the strategy **dont know choose** was applied. At the moment of setting the choice point a special *choice point mark* is pushed into Trail. If a fail occurs, the terms pushed into the Trail after the last choice point mark are freed, and are deleted (together with the choice point mark) from the Trail.

The invariant valid during the computation is the following: if we free the current resulting term and terms pushed into the Trail after the corresponding choice point mark, we will obtain the state of the memory as it was at the moment of setting of the corresponding choice point.

Let us take again the computational system from the last example and apply it to the term $f(h(a))$. The figure 1 shows the correct computation until having obtained the first result. In this figure, the reference counter (if different from 1) is displayed as the small index of the corresponding functional symbol. The figure shows a computation where one nondeterministic step (i.e. without freeing the left hand side of the rule) is provided followed by one deterministic

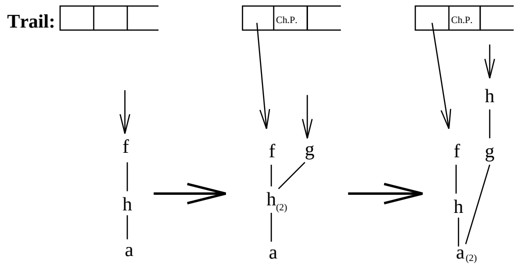


Figure 1:

application. Notice that only a needed part of the original term is copied and that the reusing of parts of terms is possible in deterministic computations (in this case in the term $g(h(a))$ we have reused the g packet to create the term $g(a))$ at its place. If we continue in the execution of this example then first the resulting term $h(g(a))$ will be freed, then (as there is no term in Trail after the choice point mark) only the choice point mark will be deleted from Trail and the evaluation will continue by the application of the rule ℓ_2 to $f(h(a))$ giving the term $h(a)$. After this the application of the rule ℓ_3 fails and all terms will be freed.

Recovering the Stack

The important part of the realization of the ELAN compiler is the implementation of the functions setChoicePoint and fail, supporting the backtracking. It is a very technical part, containing mainly the saving and recovering of machine registers and the system stack of the process.

The two functions behave similarly to the standard functions setjmp and longjmp providing long jumps. In order to better explain the problems we have met, we present first the implementation of these standard C functions. We call the *goal function* the function calling setjmp (or setChoicePoint) and the *failing function* the function calling longjmp (or fail). The *stack frame* of a function is a part of the system stack where the arguments and local variables of the function are stored. The *system stack pointer* is a machine register pointing to the top of the system stack.

The standard implementation of the long jumps (in a simplified version) is the following: the function setjmp saves the machine registers, its return address and the system stack pointer; the function longjmp recovers the saved machine registers, system stack pointer and returns to the address saved by setjmp. This implementation implies that the stack frame of the goal function has to exist at the moment of execution of longjmp. We need to eliminate this restriction as the fail in ELAN can occur in any strategy even if the strategy setting the choice point has finished (and freed its stack frame). In practice it means that (besides the job done by setjmp and longjmp) at the moment of the fail we also need to recover the system stack of the process. The naive implementation would then be the following: the function setChoicePoint saves the machine registers, its return address and the whole system stack; the function fail recovers the system stack, the machine registers and returns to the saved address. In this implementation, the saving of the whole system stack is too expensive, in the average case only a small part of the system stack needs to be changed at the moment of failure. For example, let us suppose that we have a function f_0 calling the setChoicePoint and a function f_1 calling first f_0 and then fail. An execution of f_1 creates the stack frame of f_1 in the system stack. Then f_1 calls f_0, this pushes the stack frame of f_0 onto the stack. So, when the choice point is set two stack frames (f_1 and f_0) are on the stack. After this, when leaving f_0 its stack frame is freed and execution continues in f_1 by fail. But, at this moment the stack contains the stack frame of f_1. So, to reconstitute the stack as it was at the moment of the choice only the stack frame of f_0 has to be pushed onto the current system stack.

The example illustrates the fact that it is useless to copy the whole system stack because only the stack frames of some functions are concerned. Our implementation of the setChoicePoint uses this idea. Because of the lack of the space and because the subject is very technical, we do not explain the implementation details. The main idea consists of implementing a special *handle function* which saves the top stack frame of the system stack. Return addresses of nondeterministic functions are successively (first by setChoicePoint and then by the handle function itself) modified to point to this handle function. This guarantees that the handle function is called each time a nondeterministic function executes the return instruction (and frees its stack frame). These calls save the corresponding stack frames which are then used to reconstitute the original system stack by the fail function.

The implementation is quite technical, it is done in assembler, as there is no direct access to machine registers from the C programming language. It represents about 200 lines of code and is currently working on the SPARC-architecture and on the processors Intel 80x86.

4 Comparing to the WAM

The Warren Abstract Machine (WAM) [War83] used to implement Prolog-like languages is a well known architecture providing nondeterministic computations. For this reason we have decided to compare our implementation to this abstract architecture. We are not going to compare the generated code, this would not be possible as the two implementations implement two very different languages. We provide rather a conceptual comparison of the organisation of the main data structures used during the nondeterministic search. In both architectures there are three main data structures: *Heap* is the "working memory" containing the bodies of the terms; *Stack* stores function arguments and local variables (and choice points in the WAM) and *Trail* contains some specific information used during the backtracking. An additional data structure is used in our implementation to store the choice points and the stack frames of nondeterministic functions.

Several differences between both architectures can be seen on these data structures: The Heap in the WAM is a LIFO (Last In First Out) data structure, it increases while setting the choice points and decreases at the moment of fail. In our implementation the Heap is a stable piece of the memory handled by the reference counting garbage collector.

The Stack has a very similar function in the WAM and in our implementation. Nevertheless, in our implementation the Stack is directly the system stack of the process (and the top of the Stack is obviously in machine registers). During the nondeterministic computations the freed stack frames are saved into a special additional data structure from where they are recovered at the moment of fail. This double copying is avoided in the WAM where, after the setting of a choice point the Stack does not decrease and no stack frame is freed. We did not made this optimization, because in our case this non-standard behaviour of the system stack can provoke problems when using standard C compilers with the register optimizations.

The Trail is in both cases (the WAM and the ELAN compiler) an additional LIFO data structure. The WAM's Trail contains pointers to variables which have to be un-installed when a fail occurs, in our implementation the Trail contains pointers to terms which have to be freed when a fail occurs.

5 Experimental Results

We are aware of the difficulties of trying to compare different software using experimental results which can be sensitive to the choice of computer architecture and benchmarks. In this section we present only a few experimental results to illustrate the level of efficiency which can be obtained by the presented method of compilation. Contrary to possible expectations, we do not compare our implementation to some other compilers of term rewriting. In general, these systems compile term rewriting into functional language and it seems evident that they can not supersede the efficiency of programs written directly in a functional language.

In particular, we compare here several benchmarks executed by the ELAN compiler, Standard ML compiler of New-Jersey [AM87] Version 0.93 (for deterministic benchmarks) and ECLiPSe [MS+93] Version 3.5.1 (for nondeterministic benchmarks). ECLiPSe is an ECRC implementation of a Prolog extension. The figure 2 summarises the execution times (in seconds) measured on a Sun-4 workstation for the following benchmarks: fib - the naive implementation of the n-th Fibonacci number; $nrev$ - the naive reverse of a list of size n; $primes$ - computes the list of prime numbers from 1 to n; $lrev$ - reverse in linear time of a list of size n; $perm$ - computes all permutations of a list of size n; $queens$ - the n queens on the chessboard problem. The notation $n * problem$ means that the $problem$ was executed n times and the total time of these executions was measured.

The performance of the compiler can be partially explained by the use of the assembler library for backtracking and by the reference counting garbage

$Problem$	$SML-NJ$	$ELAN$	$ECLiPSe$
$fib(33)$	8.4	8.4	
$nrev(5000)$	17.19	5.7	
$primes(50000)$	12.68	7.8	
$10 * lrev(100000)$	4.08	2.8	
$100000 * lrev(100)$	4.01	11.3	
$perm(8)$		1.8	2.86
$queens(8)$		0.4	2.25

Figure 2: Experimental results.

collector, which permits reusing parts of terms of left hand sides of rewriting rules to build right hand sides without producing garbage. The biggest advantage of these two techniques is that no data structure for storing the "living" variables is needed to be maintained for garbage collection. This allows us to store variables directly in machine registers and to obtain a speed-up due to the good register optimization in compiled programs. On the other hand reference counting needs an active procedure of freeing garbage (if any) and can then slow the execution time.

6 Summary

In this paper we have presented a compiler for the ELAN programming language. The language permits us to combine deterministic and nondeterministic computations using the formalism of term rewriting systems. The resulting implementation uses a reference counting garbage collector, and do not use any additional stack (beyond the system stack) to store function's arguments and local variables. A new technique was developed to compile the non-standard control flow during the backtracking. The main theoretical contribution of this implementation is the combination of nondeterministic search with the efficient implementation techniques of deterministic rewriting without losing efficiency.

The implementation of the compiler represents an important task, it consists of about 12000 lines of (we think, intelligent) C++ code. Beyond the few presented benchmarks, the compiler has been tested on several non-trivial tasks with satisfactory results.

Acknowledgement: Many thanks to all persons contributing to this work, in particular to Peter Borovanský, Claude and Hélène Kirchner, and Pierre-Etienne Moreau.

References

[AM87] A. W. Appel and D. B. MacQueen. A Standard ML Compiler. In *Proceedings of the Conference on Functional Programming Languages and Com-*

puter Architecture, volume 274 of *Lecture Notes in Computer Science*, pages 301–324, Portland, Oregon, USA, September 14–16, 1987. Springer-Verlag.

[Aug85] L. Augustsson. Compiling pattern matching. In *Proceedings 2nd Conference on Functional Programming Languages and Computer Architecture, Nancy (France)*, volume 201 of *Lecture Notes in Computer Science*, Nancy (France), 1985. Springer-Verlag.

[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.

[Frü95] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1995.

[HO82] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[Kap87] S. Kaplan. A compiler for conditional term rewriting system. In P. Lescanne, editor, *Proceedings 2nd Conference on Rewriting Techniques and Applications, Bordeaux (France)*, volume 256 of *Lecture Notes in Computer Science*, pages 25–41, Bordeaux (France), May 1987. Springer-Verlag.

[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.*, pages 131–158. The MIT press, 1995.

[KL91] E. Kounalis and D. Lugiez. Compilation of pattern matching with associative commutative functions. In *16th Colloquium on Trees in Algebra and Programming*, volume 493 of *Lecture Notes in Computer Science*, pages 57–73. Springer-Verlag, 1991.

[MOM93] N. Martì-Oliet and J. Meseguer. Rewriting logic as a logical and semantical framework. Technical report, SRI International, May 1993.

[MS+93] M. Meier, J. Schimpf, et al. ECLiPSe User Manual. Technical report ECRC-93-6, ECRC, Munich (Germany), 1993.

[Vit90] M. Vittek. An efficient many-to-one unification algorithm for linear terms. (in Slovak language). In *Proceedings of the First Czechoslovak Logic Programming Conference*, 1990.

[Vit94] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

[War83] D. H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, CA, October 1983.