Compiling nondeterministic computations

PEM

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

Abstract. Implementing a compiler for such a language with nondeterministic features is known to be a difficult task. This paper presents two new functions setChoicePoint and fail that extend the C language to efficiently handle choice point management. Algorithms and implementation techniques are detailed. As an application, we give compilation schemes to illustrate the power and the easy use of setChoicePoint and fail to design new compilation schemes for nondeterministic computation in the system ELAN.

1 Introduction

In the area of formal specifications, rewriting techniques have been developed for two main applications: prototyping algebraic specifications of user-defined data types and theorem proving related to program verification. In this context we are interested in nondeterministic computation and deduction.

Term rewriting is nondeterministic in the sense that there may be several reductions starting from one initial term and producing different results. Rewriting logic [Mes92] gives a logical background and raises new interesting problems concerning the efficient implementation of nondeterministic rewriting which needs backtracking. This is similar to the implementation of logic programming languages, but a significant difference is the fact that rewriting rules can be applied inside the terms. Moreover the formalism used to prune the search space is different from that of logic programming languages.

In this paper we present a new technique 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, this can be interesting to test it in implementations of constraint solvers, of the WAM [War83, AK90] and Prolog-like languages.

A first implementation has been done in 1996. The experimental results, presented in [Vit96], show that nondeterministic rewriting can be implemented as efficiently as the best current implementations of functional and logic programming languages. This paper presents a formalisation of the implementation and gives detailed algorithms to re-use, adapt and improve the proposed method.

Section 2 illustrates the behaviour of usual functions used to implement backtracking in nondeterministic computations. Section 3 gives a brief overview of existing techniques for compiling languages with nondeterministic features into C. Section 4 presents algorithms for two new proposed functions: setChoicePoint
and fail that implement an efficient backtracking control flow. Then Section 5 gives an overview of the ELAN system based on term rewriting. Some compilation schemes are presented in order to illustrate how it is smart and easy to use setChoicePoint and fail to design new compiler for languages with nondeterministic computations.

2 Basic choice point primitives

Backtracking is a well-known approach to implement nondeterministic computations. In compilation techniques, two functions are usually needed: the first one to create a choice point and save the execution environment. The second one to backtrack to the last created choice point and restore the saved environment. Many languages that offer nondeterministic capabilities provide similar functions: for instance world+ and world- in Claire [CL96], try and retry in WAM, setChoicePoint and fail in ELAN [Vit96].

We propose to extend the C language by adding two flow control functions: setChoicePoint and fail. setChoicePoint returns the integer 0 when setting a choice point, and the computation goes on. Then when the function fail is called, it performs a jump into the last call of setChoicePoint and it returns the integer 1. 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. Those two functions do not have such a limitation.

The following program, written in a pseudo-code, illustrates the behaviour of these two new functions:

<table>
<thead>
<tr>
<th>Program</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>global counter = 0;</code></td>
<td><code>result = 0, locvar = 0, counter = 0</code></td>
</tr>
<tr>
<td><code>local locvar = 0;</code></td>
<td><code>locvar = 1, counter = 1</code></td>
</tr>
<tr>
<td><code>if setChoicePoint = 1 then</code></td>
<td><code>result = 1, locvar = 0, counter = 1</code></td>
</tr>
<tr>
<td><code>end</code></td>
<td><code>locvar = 1, counter = 2</code></td>
</tr>
<tr>
<td><code>result = setChoicePoint;</code></td>
<td><code>end</code></td>
</tr>
<tr>
<td><code>print result, locvar, counter;</code></td>
<td></td>
</tr>
<tr>
<td><code>locvar +=; counter +=;</code></td>
<td></td>
</tr>
<tr>
<td><code>print locvar, counter;</code></td>
<td></td>
</tr>
<tr>
<td><code>fail;</code></td>
<td></td>
</tr>
</tbody>
</table>

`locvar` is a local variable saved by setChoicePoint while `counter` is a non-saved global variable. The variable `result` contains returned values of successive calls to setChoicePoint. The Run column shows the output obtained when executing the program: variables are initialised to 0, then a first choice point is created and the computation goes on. A second choice point is created, variables are printed, incremented and printed. Before doing the first fail, `counter = 1` and `locvar = 1`. Then a backtrack is performed: the function fail restores the last saved environment (`locvar = 0`) and transfers the control to setChoicePoint function which returns the integer 1. This explains why the third line is `result = 1, locvar = 0, counter = 1`. The function fail is called: a backtrack to the first set choice point is performed. The conditional test is evaluated to true and the program stops.
3 Known choice point implementations

The implementation of choice point management most often involves two mechanisms: an environment stack, called trail, to save local variable values and the continuation address; and a control flow handler to perform the jump to the saved continuation address when a backtrack is done.

A number of techniques for implementing branching schemes have been proposed over the years, especially in the functional and logic programming communities. Several languages use C as target language: Janus, Erlang, KL1, and Mercury, for example. Their different compilation schemes are presented in [CD95, HCS96, DM94].

The naive method implements branching using a C goto statement. However problems arise because indirect branching is not available in standard C and also because goto instruction can only do jump into its function scope. This leads to a C program composed of a unique huge function with a switch statement to simulate indirect gotos. This compilation scheme is unrealistic since it makes separate compilation impossible. Collecting all code into one C function affects compilation time and compiler’s ability to perform register allocation.

A less efficient method consists in translating each labelled block by a C function that returns continuation address. Those function are managed by a driver function that does the necessary dispatching to transfer control from one function to another. This method is not the most optimised one but is suited for standard C.

A third well-known existing scheme consists in using non standard C features that are supported by the GNU C compiler [Sta95]. The gcc compiler makes it possible to take the address of labels, and later to jump to those addresses. It also offers the possibility to insert inline assembly code, and to specify the assembly name for a function. With those extensions it is now possible to translate any branching by a goto statement.

The three presented schemes consider the program as an unstructured one. Parameter passing cannot be done in a natural way. Instead, global variables are used to communicate arguments from caller to callee. Local variables have to be saved before doing a jump: usual function calls have to be simulated. As a matter of fact, this is very difficult for a human to write a program in these conditions. Program using such nondeterministic computation can only be automatically generated.

The presented methods are efficient and well-designed to implement a WAM like abstract machine. But it is still difficult to use them to design new compilation schemes because program are usually difficult to read.

4 New choice point management

When designing new compilation schemes for a language, it is interesting to have a readable generated code. The two previously presented functions setChoicePoint and fail have been designed to compile the ELAN language into C. There is no
restriction on the generated C code: local variables and parameter passing can be used.

The idea consists in mapping the environment stack to the system stack.

In order to present two different implementations of setChoicePoint and fail, let us explain first how to design extensions of the two standard C functions setjmp and longjmp.

4.1 setJump: an extension of setjmp

The standard C library defines two low level functions setjmp and longjmp. The first one saves the current execution context (machine registers and a return address) in a jmp_buf structure. The second one can restore any stack context that was saved in a jmp_buf structure by setjmp. After the longjmp runs, the program execution continues as if the corresponding call to the setjmp function had just returned the value specified in the longjmp call. The result of longjmp is undefined if the function that made the corresponding call to the setjmp has already returned.

We propose first to extend setjmp and longjmp into setJump and longJump to suppress such undefinedness: the whole stack system (memory block between the base pointer and the stack pointer) has to be saved in a Jump_buf structure when calling setJump.

Given an integer different from 0 and a valid Jump_buf structure, the longJump function restores registers and the system stack, and then the integer parameter is returned.

Those two functions may be implemented in C: setjmp and longjmp are used to save and restore registers and memcpy is used to copy memory blocks. However, longjmp is used in a non-standard way. To get a safe implementation, setJump and longJump have to be re-implemented in assembly language. Note that setjmp and longjmp are implemented themselves in assembly language.

Some processors are based on a window register architecture (Sparc processors for example). In this case, it may not be possible to save and restore the window position, even in assembly language. In Section 4.4, we will present a new algorithm that can be implemented on such architectures.

4.2 A first implementation of setChoicePoint and fail

setChoicePoint and fail are somehow restrictions of setJump and longJump because the fail function always restores context of the last set choice point. This special case allows us to design smarter and more efficient algorithms. A naive implementation of setChoicePoint and fail consists in reusing setJump, longJump implementations and storing Jump_buf structures in a FIFO data structure (the trail itself).

Let us remark that 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 instant of failure. For example, let us consider the following program, where dots denote irrelevant instructions:
void main() { void g(int arg) {
  ...  ...
  g();  setChoicePoint();
  ...  ...
  fail();
  }
  ...
}

An execution of the main function creates the stack frame of main in the system stack. Then main calls g, this pushes the stack frame of g onto the stack. So, when the choice point is set, two stack frames (main and g) are on the stack (see Figure 1). After this, when leaving g, its stack frame is freed and execution continues in main by fail. But, at this moment, the stack contains the stack frame of main. So, to reconstitute the stack as it was at the moment of the choice, only the stack frame of g has to be restored onto the current system stack.

Fig. 1. Lazy stack saving

This example illustrates the fact that this is useless to copy the whole system stack because only the stack frames of some functions are concerned. The next implementation of the setChoicePoint uses this idea.

4.3 Notations

In order to give a detailed algorithm several notations have to be defined. They are useful to clearly compute memory blocks that have to be saved and restored.

Let us first consider a simple execution model that consists in viewing a program execution as a sequence of:

- instructions executions; and
- function calls; and
- function returns.
Let us define watch points $\tau_1, \ldots, \tau_n$, as the first executed instruction after a function call or a function return.

When running the program presented in Section 4.2, the function `main` calls `g` which calls `setChoicePoint`. The first executed instruction when entering `main`, `g` and `setChoicePoint` corresponds to watch points $\tau_1$, $\tau_2$ and $\tau_3$ respectively. The first executed instruction after `setChoicePoint` had returned, corresponds to $\tau_4$. Other watch points $\tau_5$, $\tau_6$ and $\tau_7$ are defined similarly. The program execution and watch-points placements are illustrated in Figure 2.

When executing a function, there is a corresponding environment called $\epsilon_j$ that contains information about the current executing function. This information is available through different functions:

- $fp(\epsilon_j)$ to get the frame pointer value,
- $ra(\epsilon_j)$ to get the return address value,
- local variables are also saved in an environment.

To each watch point $\tau_i$ is associated an environment $\epsilon_j$ ($i$ and $j$ are not equal in general because several watch-points may be associated to a given environment) and a current stack pointer value $sp(\tau_i)$.

Let us define $Env\cup wt$ the surjective function from $\{\tau_1, \ldots, \tau_m\}$ to $\{\epsilon_1, \ldots, \epsilon_n\}$ that maps each $\tau_i$ to its environment $\epsilon_j$. The set $\{\tau_1, \ldots, \tau_m\}$ is totally ordered by indices values: $\tau_1 < \cdots < \tau_m$.

In the previous example (see Figure 2 for illustration), we have:
Environments are organised as blocks in a block structure language and the notion of embedded environment can be defined. For a given $\epsilon_j$, the embedded environment $\text{emb}(\epsilon_j)$ is the environment in which the function associated to $\epsilon_j$ is called.

In the previous example, we have:

- $\text{env}_1(\tau_1)$ = $\text{env}_1(\tau_5)$ = $\text{env}_1(\tau_7)$ = $\epsilon_1$;
- $\text{env}_1(\tau_3)$ = $\text{env}_1(\tau_4)$ = $\epsilon_2$;
- $\text{env}_1(\tau_3)$ = $\epsilon_3$;
- $\text{env}_1(\tau_6)$ = $\epsilon_4$.

Let us define for $\epsilon \in \{\epsilon_1, \ldots, \epsilon_n\}$ the functions that returns the minimum element (resp. maximum) of the inverse image of $\text{env}_1$:

- $\text{Min}(\epsilon) = \min(\text{env}_1^{-1}(\epsilon))$
- $\text{Max}(\epsilon) = \max(\text{env}_1^{-1}(\epsilon))$

Let $<$ be a total ordering on addresses such that the base pointer $bp$ is the minimum. Let $l_1, l_2$ be two addresses of the system stack. If $l_1 < l_2$, $l_1$ is said to be nearer to $bp$ than $l_2$. If $l_1 < l_2$, $[l_1, l_2]$ is the memory block between those two addresses.

4.4 Advanced algorithm for setChoicePoint and fail

The goal of this algorithm, roughly presented in Section 4.2, is to minimise the number of saved stack frames. The main idea consists in implementing a special handle function which saves the top stack frame of the system stack.

Each time a nondeterministic function returns, the last jump is rerouted to this handle function; 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 recover the original system stack by the fail function.

The setChoicePoint algorithm contains two parts: machine register save and handle function activation.

Let $\tau_i$ be a ChoicePoint: setChoicePoint saves registers, which include $sp(\tau_i)$, $ra(\text{env}_1(\tau_i))$, and $fp(\text{env}_1(\tau_i))$, pushes the special mark $\text{endReg}$ into the trail and then jumps into the handle function: saveFrame.

Let $\epsilon_j$ be the environment of the function that did the branching to saveFrame (when creating the choice point, $\text{env}_1(\tau_i) = \epsilon_j$):
- `saveFrame` saves the frame of the function that called the function associated to \( \epsilon_j \). Let call \( emb(\epsilon_j) \) its environment. The saved frame is \( [fp(emb(\epsilon_j)), sp(Min(\epsilon_j))] \). Then `saveFrame` pushes the special mark `endFrame` into the trail.
- `saveFrame` replaces the callers' return address (saved in \( emb(\epsilon_j) \)) by the `saveFrame` procedure's address. Each time a function returns, the save frame handler is activated.

![Trail stack state after setting a choice point](image)

**Fig. 3.** Trail stack state after setting a choice point

The handle function `saveFrame` may be called several times before a `fail` occurs. In this case, several frames are saved into the trail. Figure 3 illustrates this possibility: a choice point has been created and then two frames have been saved. This occurs when the function calling `setChoicePoint` had returned. Let us notice that saved frames are linked and associated to a single choice point.

The `fail` algorithm rebuilds the system stack with saved frames until the `endReg` code is found: `fail` restores registers and the stack pointer, and then returns the integer 1.

Note that `fail` removes the last created `ChoicePoint` and restores the system stack in its initial state. Assuming that a return address is saved in the system stack, by recovering the stack, the return address (that was modified to `saveFrame`) is restored to its initial value: the save frame handler is no longer active.

### 4.5 Detailed implementation

In this section we give a low level description that corresponds to the assembly language implementation:
Algorithm 1: setChoicePoint

The following figure illustrates the state of the trail stack after executing setChoicePoint. This corresponds to the top level of the zoom part given in Figure 3.

<table>
<thead>
<tr>
<th>Register</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg1</td>
<td>saved registers</td>
</tr>
<tr>
<td></td>
<td>stack pointer</td>
</tr>
<tr>
<td>reg2</td>
<td>return address</td>
</tr>
<tr>
<td>reg2 - 3</td>
<td>caller’s frame pointer: reg0</td>
</tr>
<tr>
<td>reg2 - 2</td>
<td>initial trail pointer: reg1</td>
</tr>
<tr>
<td>reg2</td>
<td>endReg code</td>
</tr>
</tbody>
</table>

This saved information is used by the handle function saveFrame to save stack frames and update links to the corresponding choice point. Note that the notation reg2 - 3 is used to specify a base address. In this example, the value 3 is subtracted from the base register reg2 value to denote a new address whose contents is the saved return address.

The following algorithm describes in a pseudo assembly code the algorithm of the handle function saveFrame:

load the caller’s frame pointer (fp[Env.at(π)]) into reg0
load the trail pointer into reg1 and reg2
push non specific registers into the trail (referenced by reg2)
push the stack pointer (sp), the caller’s frame pointer reg0, the return address (ra) and the initial trail pointer reg1 into the trail
push the endReg code
prepare the return value: 0
do a jump to saveFrame
Algorithm 2: saveFrame

The state of the trail stack after executing saveFrame is described in Figure 3. Let us notice that top and bottom parts of the zoom stack have a similar structure: values are saved in the same order.

The following pseudo assembly code describes the fail's implementation:

Algorithm 3: fail

5 Compiling nondeterministic rewriting

Starting from the idea that inference systems can be quite conveniently described by rewrite rules, we began in the early nineties the design and implementation of a language in which inference systems can be represented in a natural way, and executed reasonably efficiently. This led us quickly to formalise such a language using the conditional rewriting logic introduced by J. Meseguer [Mes92] and to see ELAN as a logical framework where the frame logic is rewriting logic [Vit94].
5.1 Short presentation of ELAN

In ELAN, a logic can be expressed by specifying its syntax and its inference rules. The inference rules of the logic are described by conditional rewrite rules. In order to make the description executable, we introduced the notion of strategy [BKK+96].

To summarise, ELAN provides the following main features:

- A semantics based on many-sorted rewriting logic,
- A powerful language to express strategies of rewrite rule application, including don't-care and don't-know choices on strategies,
- A general pre-processor making easier the translation of a logic into rewriting logic,
- A standard library to facilitate user developments,
- Modular constructions via local or global importations as well as parametric modules,
- A generic mixfix and user-definable syntax,
- Associative commutative (AC) operators,
- A very efficient compiler for ELAN programs.

5.2 Rewrite rule

In ELAN, a rewrite rule is a pair of terms with some conditions and local affectations:

\[
[ \text{rule name} ] \quad < \text{left-hand side term} > \quad \Rightarrow \quad < \text{right-hand side term} > \quad \left( \begin{array}{c}
\text{if} < \text{boolean term} > \\
\text{where} < \text{variable} > := \text{(strategy)} < \text{term} >
\end{array} \right)
\]

Applying a rule on a term consists in matching the left-hand side then verifying that conditions are satisfied and evaluating local affectations. A special backtracking mechanism exists: if a condition is not satisfiable or a local evaluation leads to a failure, the previous nondeterministic where (if there is one) has to select another solution. If there is no way to satisfy conditions and compute local affection, the rule cannot be applied.

5.3 Strategies

Strategies is one of the main originality of ELAN. In practice, a strategy is a way to describe which computations the user is interested in. The application of a strategy to a term results in the (possibly empty) set of all terms that can be derived from the starting term using this strategy [BKK+96]. When a strategy returns an empty set of terms, we say that it fails.

The application of a rewrite rule in ELAN yields, in general, several results: i.e., there are several ways to apply a given conditional rule with local affectations. This is first due to equational matching (currently only Associative and Commutative matching) and second to the where assignment, since it may itself recursively return several possible assignments for variables, due to the use of strategies.
Thus the language provides a way to handle this non-determinism. This is done using the basic strategy operators: dc (don't care choose) and dk (don't know choose).

A local affectation is said to be nondeterministic if the top strategy operator is a dk. Similarly, a rule application is said to be nondeterministic if the rule is used in a dk strategy.

For a labelled rewrite rule \( t \), the strategy \( dc(t) \) returns \textit{at most one} result which is undeterministically taken among the possible results of the application of the rule. In practice, the current implementation returns the first one.

On the contrary, if the \( t \) rule is applied using the \( dk(t) \) strategy, then \textit{all} possible results are computed and returned by the strategy. The implementation handles these several results by an appropriate back-chaining operation.

This is extended to the application of several rules: the dk strategy results in the application of all substrategies and yields the union of all results; the application of the dc strategy returns the set of results of the first non-failing. If all sub-strategies fail, then it fails too, i.e., it yields the empty set.

Two strategies can be concatenated: this means that the second strategy will be applied on all results of the first one. In order to allow the automatic concatenation of the same strategy, ELAN offers the two iterators iterate and repeat. The strategy iterate corresponds to applying zero, then one, then two, \ldots, \( n \) times the strategy to the starting term, until the strategy fails. Thus \( \text{iterate}(s) t \) returns \( \bigcup_{n=0}^{\infty} (s^n) t \). Note that iterate returns the results one by one even when an infinite derivation exists. The strategy repeat iterates the strategy until it fails and return just the terms resulting of the last unfailing call of the strategy. It can thus be defined as \( \text{repeat}(s) t = (s^*) t \) where \( (s^* t) \) t fails.

In order to illustrate how strategies work, let us consider the example consisting of the extraction of the constituents of a list:

```
rules for elem
  e : elem;
  l : neilst;

bodies
  [extract] element(e) => e   end
  [extract] element(e,l) => e   end
  [extract] element(e,l) => element(l) end
end
```

If we assume furthermore that the constants \( a, b, c \) are of sort elem, then:

- \( \text{repeat}(\text{dk(extract)}) \) \( (a,b,c) \)
  yields the set \{a,b,c\},
- \( \text{iterate}(\text{dk(extract)}) \) \( (a,b,c) \)
  yields the set \{element(a,b),a,element(b,c),b,element(c),c\},
- \( \text{repeat}(\text{dc(extract)}) \) \( (a,b,c) \)
  yields the set \{a\}.

5.4 Compilation of strategies

The compiler transforms each strategy \( s \) into a \( C \) function \texttt{str_s} with one argument. An application of the strategy \( s \) to the term \( t \) is compiled into a call
to \texttt{str\_s}, whose argument is a pointer to \texttt{t}. When the strategy is deterministic, i.e. has only one result, the function returns a pointer to the resulting term. Otherwise, backtracking is needed to implement nondeterministic computations.

For our compilation purpose, two new flow control functions, \texttt{setChoicePoint} and \texttt{fail}, have been implemented. Let us illustrate their use on a few compilation schemes of nondeterministic computations in ELAN.

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

**Non-deterministic rule application.** This strategy tries all rules and returns all possible results; local affectations and conditions have to be evaluated with the special back-chaining convention.

For each rule, a choice point is created and matching, checking conditions and local affectations are attempted. If there is a failure in one of these steps, the choice point is removed and the next rule is tried. If the rule is applied, the choice point is kept for further backtracking. When the control comes back to the initial choice point, all possible results have been returned. This compilation scheme generalises in a straightforward way to the compilation of non-deterministic application of strategies in a \texttt{dk(S_1, \ldots, S_n)}.

Let us remind that the “\texttt{if setChoicePoint=1 then}” statement means “if the control comes from a \texttt{fail}, delete the \texttt{ChoicePoint} and do the \texttt{then} part”.

\begin{algorithm}
\begin{verbatim}
if setChoicePoint=1 then
  try next rule
end if
compilation of conditions and local affectations
right-hand side construction
\end{verbatim}
\end{algorithm}

**Algorithm 4: \texttt{dk(r_1, \ldots, r_n)}**

**Non-deterministic strategy application.** The previous compilation scheme is generalised in a straightforward way to the compilation of non-deterministic application of strategies in a \texttt{dk(S_1, \ldots, S_n)}.

\begin{algorithm}
\begin{verbatim}
for all i such that 1 \leq i < n do
  if setChoicePoint=0 then
    compilation of \texttt{S_i}
    exit
  end if
end for
the last sub-strategy \texttt{S_n} is simply compiled
\end{verbatim}
\end{algorithm}

**Algorithm 5: \texttt{dk(S_1, \ldots, S_n)}**
Let us remind that “if setChoicePoint=0 then” means “if the control does not come from a fail, create a ChoicePoint and do the then part”.

All sub-strategies are compiled into successive blocks. Strategies are tried up to find the first successful \( S_i \) and then the computation is finished. When coming from a fail, the computation goes on with the next sub-strategy \( S_{i+1} \). The last sub-strategy \( S_n \) is compiled separately: no choice point has to be created.

Non-deterministic iterator. Iterating a strategy has a simple compilation scheme.

```plaintext
loop
  if setChoicePoint=0 then
    break
  end if
  compilation of S
end loop
```

**Algorithm 6: iterate(S)**

In an infinite loop, a choice point is created and an exit from the loop (the break statement) is done. The current term is returned. So applying iterate\((S)\) on a term \( t \) returns \( t \) first (zero application of \( S \)). When coming back with a failure which means that the computation of the \( n \)-th iteration is finished. If the strategy \( S \) can be applied once on the result, a new choice point is created and \( S^{n+1}(t) \) is returned. Else, the iterate strategy is finished and a backtrack is performed.

Deterministic iterator. repeat differs from iterate because only the last result is returned:

```plaintext
lastTerm = subject
if setChoicePoint=0 then
  loop
    lastTerm = compilation of S
  end loop
  end if
```

**Algorithm 7: repeat(S)**

Each time the strategy \( S \) is applied, the resulting term is saved in a special variable \( lastTerm \). When a fail is executed, the saved term \( lastTerm \) is returned.

6 Concluding Remarks

The design and the implementation of the ELAN compiler has shown in practice the usefulness of the presented functions: setChoicePoint and fail. Their “plug-in” design makes them easy to use: conventional programming techniques such
as function calls, local variables, parameters passing, modular compilation are compatible with the proposed C language extension.

When designing compilation schemes for a new language, the backtracking management, if any, is always a difficult task to solve. The proposed approach seems to be a good compromise between simplicity and efficiency: the use of high level concepts such as choice points and backtrack and a low level implementation (in assembly) to get good performances.

Experimental results show that the proposed implementation is comparable with a had-hoc approach like one used in optimised WAM implementations. For instance, the 8-queens benchmark runs 5 times faster in ELAN than in ECLiPSe [MS+93] (an ECRC implementation of a Prolog extension).

References


This article was processed using the \LaTeX\ macro package with LLNCS style.