A choice-point library for backtrack programming

Pierre-Etienne Moreau
LORIA-CNRS & INRIA-Lorraine
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France
E-mail: Pierre-Etienne.Moreau@loria.fr

Abstract

Implementing a compiler for 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. Originally, these two functions were designed to compile the ELAN strategy language. However, they can be used by programmers for general programming in C. We illustrate their use by presenting the classical 8-queens problem and giving some experimental results. Algorithms and implementation techniques are sufficiently detailed to be easily modified and re-implemented.

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 [12] 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 is of more general interest; it could be used in implementations of constraint solvers, imperative languages with backtracking such as Alma-0 [14, 2], the WAM [18, 1] and Prolog-like languages.

A first implementation of our techniques has been done by Marian Vittek in 1996. The experimental results, presented in [16], 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. It took great benefit from the idea and comments of Marian Vittek.

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 implementing choice points and compiling languages with nondeterministic features into C. Section 4 presents algorithms for the two new proposed functions that implement an efficient backtracking control flow: setChoicePoint and fail. Then Section 5 illustrates on one example how the use of setChoicePoint and fail can help in solving in a natural way algorithms that involve search. Some experimental results show that the proposed method can be a good alternative to compile in an efficient way languages that involve nondeterministic features.

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 [6], try and retry in the WAM, onfail, fail, createlog and replaylog in the Alma-0 Abstract Machine [14], setChoicePoint and fail in ELAN [16]. Recently, a new approach to the implementation of tabling for prolog [9] has been proposed. The authors suggest to extend a prolog implementation by adding some new built-in predicates. For their purpose, a similar idea as the one presented here has been explored in a different context.

We propose to extend the C language by adding two control flow functions: setChoicePoint and fail. setChoicePoint returns the integer 0 when setting a choice point, and the computation goes on. 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`. Functions setChoicePoint and fail do not have such a limitation.

The following program, written in C, illustrates the behaviour of these two new functions. The **Output** column shows the result obtained when executing the program:

Program	**Output**
`static int counter=0;`	`result=0, locvar=0, counter=0`
`main() {`	`locvar=1, counter=1`
`  if(setChoicePoint()!=0) exit(0);`	`result=1, locvar=0, counter=1`
`  f();`	`locvar=1, counter=2`
`  fail();`	
`}`	
`f() {`	
`  int result, locvar=0;`	
`  result=setChoicePoint();`	
`  printf(result,locvar,counter);`	
`  locvar++; counter++;`	
`  printf(locvar,counter);`	
`}`	

When setting a choice point, only local variables are saved. If a failure occurs, only local variables are restored to the value they had when setting the choice point. When executing the example program, a first choice point is created and the computation goes on. The function `f()` is called and `locvar` is initialised to 0. Then a second choice point is created, `result` and `locvar` are saved, printed, incremented and printed again. Before executing the first fail, `counter=1` and `locvar=1`. Then a backtrack is performed: the function fail restores the last saved environment (`f` is re-activated, `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 again: 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: first, an environment stack, called the *trail*, to save local variable values and a continuation address; second, a control flow handler to perform the jump to the saved continuation address when backtracking.

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 such as Cg, Icon, Janus, Erlang, KL1, and Mercury. Their different compilation schemes are presented in [5, 17, 8, 7, 10].

Among these techniques, the simplest method implements branching using a C `goto` statement. However problems arise because indirect branching is not available in standard C and also because a `goto` instruction can only do a 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 impossible separate compilation. Moreover, collecting all codes into one C function affects compilation time and compiler's ability to perform register allocation.

The second method consists in translating each labelled block by a C function that returns a continuation address. Those functions are managed by a driver function that does the necessary dispatching to transfer control from one function to another. Consequently, this method is not the most optimised one but is suited for standard C and separate compilation.

A third well-known existing scheme consists in using non standard C features that are supported by the GNU C compiler [15]. The `gcc` compiler makes it possible to take the address of labels, and later on to jump to those addresses. It also offers the possibility to insert inline assembly code, and to specify the assembly name of a function. With those extensions it is now possible to translate any branching by a `goto` statement.

When using one of the three presented schemes, the structure of a program is not taken into account. Parameter passing cannot be done in a natural way. Instead, global variables are used to communicate arguments from caller to callee. That is why local variables have to be saved before doing a jump and classical function calls have to be simulated. As a matter of fact, it is very difficult for a human to write a program in these conditions and the presented compilation schemes can only be used in automatically generated programs.

Even if the three presented methods are efficient and well-designed to implement a WAM-like abstract machine, it is still difficult to use them to design new compilation schemes because resulting programs are often difficult to read.

4 New choice point management

In this section we present algorithms and implementation techniques of the two new functions setChoicePoint and fail.

The key idea of our approach is to use the system stack and only one environment stack to store values that have to be saved. Consequently, there is no restriction on the usage of local variables and parameter passing when programming with setChoicePoint and fail.

We first present an approach which consists in extending the two standard C functions `setjmp` and `longjmp`. Then we define some notations in order to give detailed algorithms of the second method which minimises the size of memory blocks that have to be saved (resp. restored) when setting a choice point (resp. performing a failure).

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 has been 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 whole system stack, and then the integer parameter is returned.

Depending on the architecture, these two functions may be implemented in C: `setjmp` and `longjmp` are used to save and restore registers[1] and `memcpy` is used to copy memory blocks. This approach was successful on a PC under Linux and a DEC Alpha-Station but the result is not really safe: as mentioned previously, `longjmp` is used in a non-standard way; since in is not possible in C to get the *base pointer* and the *stack pointer*, some heuristics are used; and furthermore, the behaviour is not stable when using advanced optimisation options such as `gcc -O6`. This why we recommend this approach only to get a first easy implementation, but to get a safe implementation, setJump and longJump have to be re-implemented in assembly language. There is nothing surprising, because `setjmp` are `longjmp` are themselves implemented in assembly language.

Some processors, such as Sparc, are based on a window register architecture. In this case, it is not possible to save and restore the window position: the corresponding assembly instruction must be executed in privileged mode (not accessible by users). This restriction makes impossible the implementation of setJump and longJump on such architecture, however, in Section 4.4, we present a general algorithm for setChoicePoint and fail that can be implemented on almost any 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 the 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 LIFO data structure (the *trail* itself).

Let us remark that saving the whole system stack is too expensive. In average, only a small part of the system stack needs to be changed when a failure occurs. For example, let us consider the following program, where dots denote irrelevant instructions:

[1] a similar idea is used in the BDW Garbage Collector [3]

```
void main() {               void g(int arg) {
  ...                         ...
  g(i);                       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, only the stack frame of `g` has to be restored onto the current system stack to reconstitute the stack as it was at the moment of the choice.

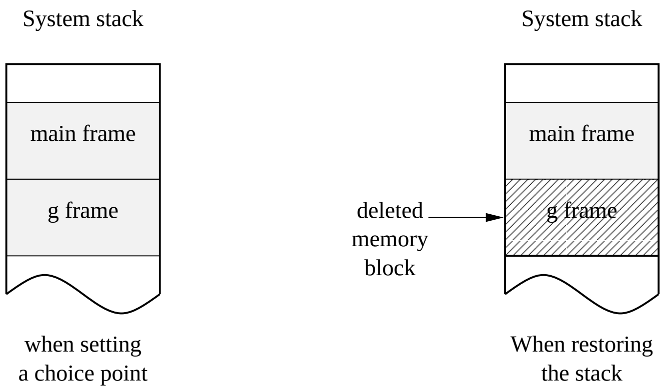

Figure 1: It is useless to copy the whole system stack

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 setChoicePoint uses this idea.

4.3 Notations

In order to give a detailed algorithm several notations are needed. 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 instruction executions, function calls, and function returns.

Let us define watch points $\tau_1, \ldots, \tau_{n_\tau}$ 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 τ_1, τ_2 and τ_3 respectively. The first executed instruction after setChoicePoint corresponds to τ_4. Other watch points τ_5, τ_6 and τ_7 are defined similarly. The program execution and watch-points are illustrated in Figure 2.

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

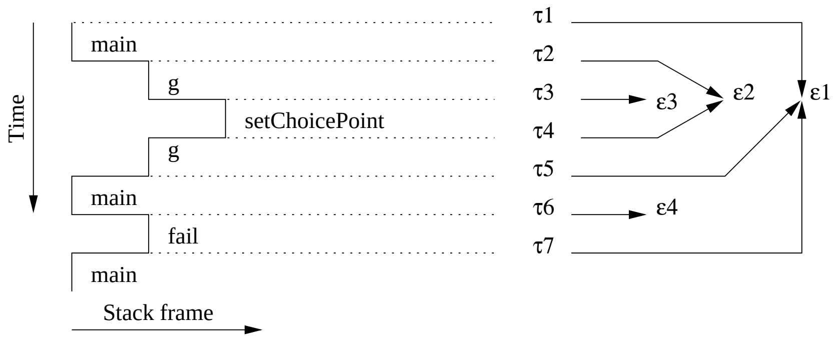

Figure 2: Setting watch points and environments

- $fp(\epsilon_j)$ to get the frame pointer value, i.e. the address which indicates the beginning of the stack frame;
- $ra(\epsilon_j)$ to get the return address value, i.e. the address from the program to which the program counter should be restored;
- local variables are also saved in an environment.

To each watch point τ_i is associated an environment ϵ_j (i and j are not equal in general because several watch-points may be associated to a given environment). The notion of environment and the correspondences between (τ_i) and (ϵ_j) are illustrated in Figure 2.

A stack pointer value $sp(\tau_i)$, which is the current top of the system stack, is associated to each watch point τ_i.

Let us define Env_at as the surjective function from $\{\tau_1, \dots, \tau_m\}$ to $\{\epsilon_1, \dots, \epsilon_n\}$ that maps each (τ_i) to its environment (ϵ_j). The set $\{\tau_1, \dots, \tau_m\}$ is totally ordered by indices values: $\tau_1 < \dots < \tau_m$.

In the previous example we have:

- $Env\_at(\tau_1) = Env\_at(\tau_5) = Env\_at(\tau_7) = \epsilon_1$;
- $Env\_at(\tau_2) = Env\_at(\tau_4) = \epsilon_2$;
- $Env\_at(\tau_3) = \epsilon_3$;
- $Env\_at(\tau_6) = \epsilon_4$.

Environments are organised as in a block structure language, thus, the notion of *embedded* environment can be defined: for a given ϵ_j, the embedded environment $emb(\epsilon_j)$ is the environment in which the function associated to ϵ_j is called.

In the previous example, we have:

- $emb(\epsilon_3) = \epsilon_2$
- $emb(\epsilon_2) = emb(\epsilon_4) = \epsilon_1$
- $emb(\epsilon_1)$ is not defined

Let us define for $\epsilon \in \{\epsilon_1, \ldots, \epsilon_n\}$ the function that returns the minimum element of the inverse image of Env_at: $Min(\epsilon) = min(Env\_at^{-1}(\epsilon))$

From a practical point of view, τ_i is a program's address that contains an assembly instruction. ϵ_j is a stack frame associated to the current executed C function and $Min(\epsilon)$ is the first executed instruction when entering a C function.

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 closer to bp than l_2. When $l_1 < l_2$, $[l_1, l_2[$ is the memory block between those two addresses.

4.4 Advanced algorithm for setChoicePoint and fail

The goal of the algorithm sketched 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[2] returns to the caller function, the last jump is redirected 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. These calls save the corresponding stack frames which are then used to recover the original system stack by the fail function. The setChoicePoint algorithm performs two actions: save machine registers and activate the handle function.

Let τ_i be a choice point: setChoicePoint saves registers, which include $ra(Env\_at(\tau_i))$, $sp(\tau_i)$ and $fp(Env\_at(\tau_i))$, pushes the special mark endReg into the trail and then jumps into the handle function: saveFrame.

Let ϵ_j be the environment of the function which did the branching to saveFrame (when creating the choice point, $Env\_at(\tau_i) = \epsilon_j$):

- saveFrame saves the frame of the function that called the function associated to ϵ_j. Let us 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 caller's return address (saved in $emb(\epsilon_j)$) by the saveFrame procedure's address. Thus, each time a function returns, the save frame handler is activated.

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: two choice points have been created and three frames have been saved (two of them are associated to the first choice point). This situation occurs when the function calling setChoicePoint had returned.

The fail algorithm rebuilds the system stack with saved frames until the endReg code is found. Then, the registers and the stack pointer are restored, and the integer 1 is returned.

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

[2]a function is said to be nondeterministic if a choice point is set or a nondeterministic function is called during its execution

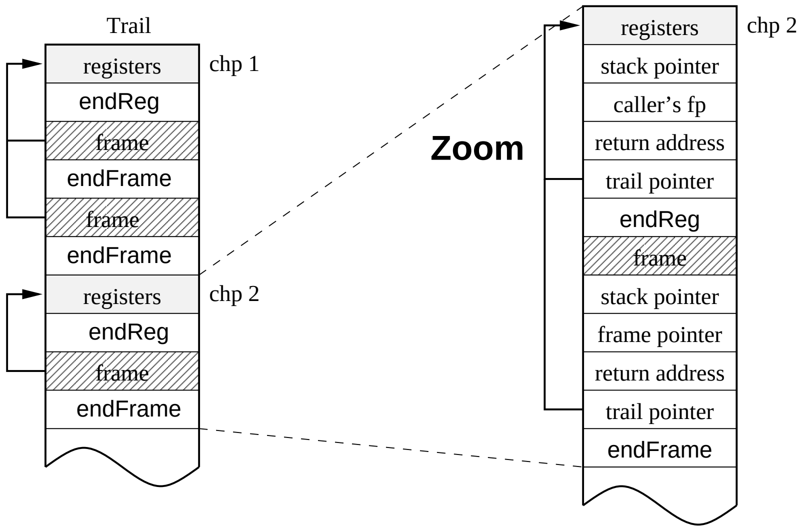

Figure 3: Trail stack state after setting a choice point

4.5 Detailed implementation

In this section we give a low level description that corresponds to the assembly language implementation. The next algorithm describes the implementation of setChoicePoint:

Algorithm 1 setChoicePoint

load the caller's frame pointer ($fp(Env\_at(\tau_i))$) into reg_0
load the trail pointer into reg_1 and reg_2
push non specific registers into the trail (referenced by reg_2)
push the stack pointer (sp), the caller's frame pointer reg_0, the return address (ra) and the initial trail pointer reg_1 into the trail
push the endReg code
prepare the return value: 0
do a jump to saveFrame

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.

reg_1	$\rightarrow$	... saved registers ...
		stack pointer
		caller's frame pointer: reg_0
$reg_2 - 3$	$\rightarrow$	return address
$reg_2 - 2$	$\rightarrow$	initial trail pointer: reg_1
		endReg code
reg_2	$\rightarrow$	

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 $reg_2 - 3$ is used to specify a base address. In this example, the value 3 is subtracted from the base register reg_2 value to denote a new address whose contents is the saved return address.

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

Algorithm 2 saveFrame

```
load the trail pointer into reg_2
load the caller's frame pointer into reg_0
restore the saved return address (reg_2 - 3) into reg_4
restore the saved trail pointer (reg_2 - 2) into reg_5
push the frame (memory block [reg_0, sp[) into the trail
push the stack pointer and caller's frame pointer reg_0
load the frame pointer of the previous memory block (reg_5 - 4) into reg_3
if reg_3 = reg_0 then
  {the return address is already into the trail. The link to the previous block has to be followed}
  load the old return address (reg_5 - 3) into reg_4
  load the beginning of the previous block (reg_5 - 2) into reg_5
end if
push reg_5 and reg_4 and the endFrame code into the trail
update the trail pointer
modify the caller's return address to the save frame handler 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 similar structures: values are saved in the same order.

The following pseudo assembly code describes the implementation of fail:

Algorithm 3 fail

```
load the trail pointer into reg_2
recoverFrame: load the current code (reg_2 - 1)
if it is a endReg code then
  branch to returnInCP
end if
restore the stack pointer (reg_2 - 5)
restore the frame pointer (reg_2 - 4)
compute the saved frame size and restore it
branch to recoverFrame
returnInCP: update the trail pointer
restore saved registers, including the return address register
return the value 1
```

4.6 Portability

The two functions setChoicePoint and fail are implemented with a 200 lines assembly library. This could be a source of difficulty for portability. However, in practice, the library can be easily implemented on a new architecture. In fact, only three specific access functions are required: $fp()$, $sp()$ and $ra()$ which return respectively the current frame pointer, the current stack pointer and the return address. The nondeterministic library[3] has been implemented on three different architectures: Sparc, Intel and DEC-Alpha. It has not been ported yet to HP-PA or MIPS processors, but it should not be a problem even if the stack grows from low to high addresses.

It is well-known that efficient implementations of nondeterministic library usually take benefit of gcc extensions and use inline statements to add assembly labels in the generated code. Therefore, our approach is not less portable than classical implementations of choice point managements.

5 Imperative programming with backtracking

Several languages such as 2LP [11] and Alma-0 [14, 2] have been developed to combine advantages of logic and imperative programming in order to deal in a natural way with algorithmic problems that involve search. Alma-0 extends imperative programming with some features inspired by the logic programming paradigm. For instance:

- use of boolean expressions as statements and vice versa;
- the ORELSE statement starts by proceeding through the first branch. If the computation eventually fails, backtracking takes place and the computation resumes with the next branch in the state in which the previous branch was entered;
- the SOME statement extends the ORELSE construction to generate a number of choice points determined only at run-time;
- the FORALL statement introduces a controlled form of iteration over the backtracking.

With these extensions, the program that computes all solutions of the N-queens problem can be expressed in a natural way. Let us remind that the N-queens problem consists in placing N queens on the chess board so that they do not attack each other.

```
FOR column := 1 TO N DO
  SOME row := 1 TO N DO
    FOR i := 1 TO column-1 DO
      x[i] <> row;
      x[i] <> row+column-i;
      x[i] <> row+i-column
    END;
    x[column] = row
  END
END
```

```
BEGIN
(* print all solutions *)
  nrSols := 0;
  FORALL queens(b);
    DO
      (* print solution: b *)
    END;
END queens.
```

In the Alma-0 environment, the program is first translated into the Alma-0 abstract machine language (AAA), and then, each AAA instruction is compiled into C statements. The used backtracking mechanism corresponds to the first method described in Section 3. Thus, modular compilation is not possible and optimisations performed by the C compiler are not optimal.

We think that setChoicePoint and fail could be a good alternative to implement AAA backtracking operations. Even more, setChoicePoint and fail seem to be well-suited to compile directly an Alma-0 program to C without any intermediate abstract machine. Indeed, the previous Alma-0 program fragment can be easily translated in C extended with our two functions setChoicePoint

[3]available at http://www.loria.fr/equipes/protheo/PROJECTS/ELAN

and fail. The SOME statement is replaced by the call of the from1ton function and the boolean expressions are translated into conditions and failures.

```
void main() {
 backTrackInit();
 queens(12);
 fail();
}

int from1ton(int n) {
 int i;
 for(i=1;i<n;i++) {
  if(setChoicePoint()==0) {
   /* set a choice point
      and return i */
   return(i);
  }
 }
 return(i);
}
```

```
queens(int n) {
 int col,row,i;
 for(col=1;col<=n;col++) {
  row=from1ton(n);
  for(i=1;i<col;i++) {
   if(array[i]==row ||
      array[i]==row+col-i ||
      array[i]==row+i-col) {
       /* choose another row */
       fail();
   }
  }
  array[col] = row;
 }
 /* print solution: array[] */
}
```

Remark that the library is initialised by a call to the backTrackInit function which creates an initial choice-point in order to prevent using fail before setChoicePoint.

In order to show the potential of the approach, the N-queens solving procedure has been implemented in several languages: standard C, C extended with setChoicePoint and fail, Mercury, Wamcc, Alma-0 and 2LP. In each case, even if it is not the best one, the same searching strategy has been used: a position for the current queen is chosen and is not removed from the list of remaining positions. The experimental results presented in Figure 4 show that the presented approach is as efficient as Mercury. But, you should not deduce that replacing its backtracking management by the proposed library should improve the efficiency. We can only deduce, from these benchmarks, that the proposed approach is a good compromise between simplicity and efficiency.

Let us mention that the use of setChoicePoint and fail is not restricted to the implementation of toy examples such as the N-queens problem: they are intensively used in a more general context related to the compilation of rewriting systems with strategies. In this case, they are frequently used in generated programs that may consist of several thousands lines of C code.

6 Concluding Remarks

The previous section has shown that setChoicePoint and fail are designed to extend the C language in order to deal in a natural way with problems that involve search. setChoicePoint and fail can also be used to compile languages with some features inspired by the logic programming paradigm. This is a good alternative to avoid using an abstract machine and to get interesting performances. The two functions were originally designed to compile the ELAN language. Some examples of compilation schemes are given in [13] (see appendix A).

Their "plug-in" designs make 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. Having a readable code is a key point to be able to design new complex compilation schemes.

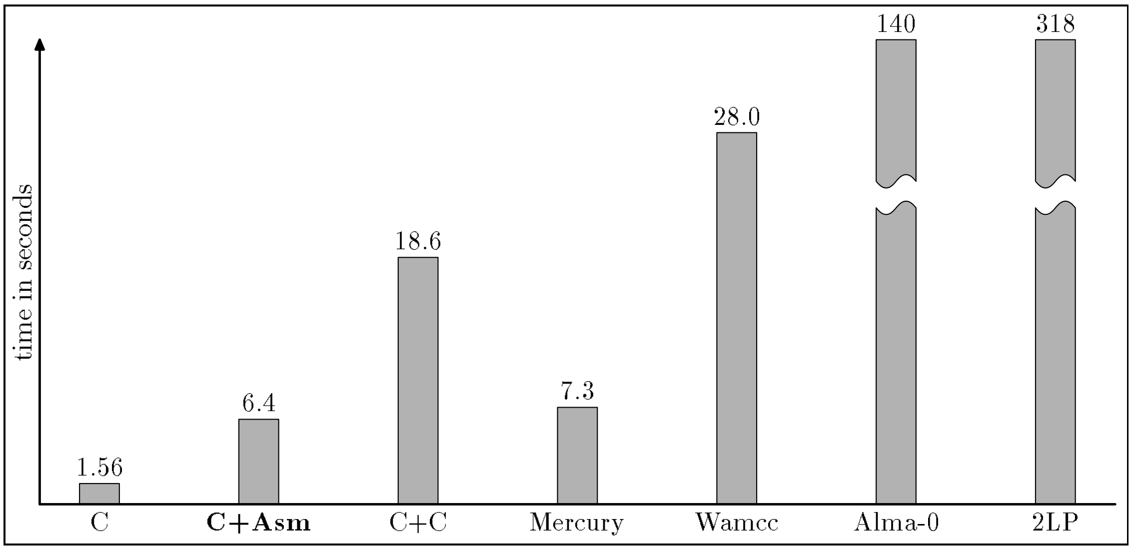

Figure 4: Several implementations of the same algorithm are compared in order to show the potential of the approach. The fastest implementation of the N-queens solving procedure is implemented in standard C with a recursive function call. Two others implementations have been done in C extended with setChoicePoint and fail: the first one `C+Asm` use the assembly version of the proposed library and the second one `C+C` use the C version presented in 4.1.

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: it provides both high level concepts such as choice points and backtracking, and a low level implementation (in assembly) to get good performances. Experimental results show that the proposed implementation is comparable with an ad-hoc approach like the one used in optimised WAM implementations.

Acknowledgements

I would like to thank the Protheo group and especially Peter Borovanský, Claude Kirchner, Hélène Kirchner and Laurent Vigneron for helpful discussions and comments. I would especially like to thank Marian Vittek and Bart Demoen for their support and enlightening conversations.

References

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

[2] K. R. Apt and A. Schaerf. Search and imperative programming. In *24th POPL*, pages 67–79, 1997.

[3] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), Sept. 1996.

[5] T. Budd. An implementation of generators in C. *Computer Languages*, 7:69–87, 1982.

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

[7] P. Codognet and D. Diaz. wamcc : Compiling Prolog to C. In *Proceedings of International Conference on Logic Programming*, Tokyo, Japan, 1995. The MIT press.

[8] B. Demoen and G. Maris. A comparison of some schemes for translating logic to C. In *Workshop on Implementations of the 11th International Conference of Logic Programming*, Santa Margherita, Italy, 1994. MIT Press.

[9] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. Personal Communication, 1998.

[10] 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.

[11] K. McAloon and C. Tretkoff. 2LP: Linear programming and logic programming. In P. Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 101–116. The MIT press, 1995.

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

[13] P.-E. Moreau. Compiling nondeterministic computations. Technical Report 98-R-005, CRIN, 1998.

[14] 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.

[15] R. Stallman. *Using and porting the GNU CC compiler*, 1995.

[16] 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.

[17] S. Wampler and R. Griswold. The implementation of generators and Goal-Directed Evaluation in Icon. *Software-Practice and Experience*, 13:495–518, 1983.

[18] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Artificial Intelligence Center, 1983.

A Appendix: Compiling nondeterministic rewriting

A.1 Short presentation of ELAN

Originally setChoicePoint and fail were designed to compile the ELAN language. In ELAN, a logic can be expressed by specifying its syntax and its inference rules described by conditional rewrite rules. In order to control the execution, the notion of strategy [4] has been introduced.

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 dont-care and dont-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; some modular constructions via local or global importations as well as parametric modules; a generic mixfix and user-definable syntax; associative commutative (AC) operators; and a very efficient compiler for ELAN programs.

A.2 Rewrite rule

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

[rule name]   < left-hand side term > ⇒ < right-hand side term >
(**if** < boolean term >
| **where** < variable > **:=** (strategy) < term >)*

Applying a rule on a term consists in matching the left-hand side, then verifying that conditions are satisfied and evaluating where statements. A special backtracking mechanism exists: if a condition is not satisfied or a local assignment 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 assignments, the rule cannot be applied.

A.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. When a strategy returns an empty set of terms, we say that it *fails*.

Thus the language provides a way to handle this non-determinism. This is done using the basic strategy operators: dc (dont care choose) and dk (dont know choose). For a labelled rewrite rule ℓ, the strategy dk(ℓ) returns *all* possible results. The implementation handles these several results by an appropriate back-chaining operation.

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,..., n times the strategy to the starting term, until the strategy fails. Thus (iterate(s))t returns $\bigcup_{n=0}^{\infty}(s^n)t$. The strategy repeat iterates the strategy until it fails and returns only the terms resulting of the last unfailing call of the strategy.

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 : nelist;
bodies
   [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:

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

A.4 Compilation of strategies

The ELAN compiler transforms each strategy s into a C function `str_s` with one argument. An application of the strategy s to the term t is compiled into a call to `str_s`, whose argument is a pointer to t. When the strategy is deterministic, i.e. has at most one result, the function returns a pointer to the resulting term. Otherwise, backtracking is needed to implement nondeterministic computations.

Let us illustrate the use of setChoicePoint and fail on a few compilation schemes of nondeterministic computations in ELAN.

Any computation in ELAN begins with a call to setChoicePoint to set an initial choice point and ends with a call to the 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.

A.4.1 Nondeterministic rule application.

This strategy tries all rules and returns all possible results; local assignments 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 nondeterministic application of strategies in a dk$(S_1, \ldots, S_n)$.

Let us remind that the "**if** setChoicePoint=1 **then**" statement means "if the control comes from a fail, delete the ChoicePoint and do the *then* part".

Algorithm 4 dk$(r_1, \ldots, r_n)$

```
if setChoicePoint=1 then
  try next rule
end if
compilation of conditions and local affectations
right-hand side construction
```

A.4.2 Nondeterministic iterator.

Iterating a strategy has a simple compilation scheme. 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 $S(S^n)(t)$ is returned. Else the failure is propagated.

Algorithm 5 iterate(S)

```
loop
  if setChoicePoint=0 then
    break
  end if
  code of S
end loop
```
