Recursion vs. Iteration in Prolog

Micha Meier

ECRC-ECRC-95-13

technical report ECRC-ECRC-95-13

Recursion vs. Iteration in Prolog

Micha Meier

European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : michaecrc.de

Abstract

We outline methods of compiling recursive procedures in Prolog to yield code which is close to that of imperative languages. Recursion is compiled into iteration which keeps loop invariants unchanged and uses destructive assignment on the stack variables. We also adentify various recursion types, depending on the stack frame type which is involved, namely environments, choice points and suspended goals.

1 Introduction

Prolog, like any other programming language, usually spends most of its execution time in a small portion of the program code, executing loops. It does not have the loop constructs heavily used in procedural languages, e.g. *while* or *for*, repetitive actions and loops (without side effects) are expressed using recursive procedures. Today, when it seems more and more likely that Prolog will reach a speed not much different from imperative languages, we have to reconsider the way we treat loops in Prolog. Although exploiting *tail recursion optimization* is usual in WAM [5], this is not enough because there is still a significant overhead on the execution time even if the recursive procedures use constant space.

Ideally, it should be possible to compile loops in Prolog so that we obtain code which is very close to that one generated by imperative languages, because this is the only way to achieve competitive speed on traditional hardware.

This paper presents an overview of the main problems in compiling recursive procedures into the WAM, and hints how these problems could be solved. In the section 2 we illustrate the problem on an example, and generalize it by extending to different data types. In the next sections we treat each of the data type, namely environments, choice points and suspended goals. In the last section we discuss the results and outline future work.

2 Iteration and Recursion

We first present a simple example to illustrate how tail-recursive procedures are normally compiled into the WAM, and what are the drawbacks compared to an imperative language, represented by C. We do not try here to optimize any other Prolog features, especially we do not try to use registers for permanent variables, but we sometimes assume that the compiler has some knowledge that can be obtained only from the global analysis of the program. We assume that the arithmetic and other simple built-in predicates are compiled in-line and that they do not require an environment creation. For simplicity we often assume that the procedure is always called with arguments of correct type, as is the case in most imperative languages and that its arguments are dereferenced. Completing the code to include these features is obvious. We assume a thorough knowledge of the WAM throughout the whole paper.

When listing the code generated by the compiler, we deliberately mix WAM code, destructive assignment expressed by the $:=$ operator, C-like syntax for expressions which cannot be expressed in the WAM like e.g. if-then-else, and macros with understandable mnemonic names. To save space, we sometimes do not list complete sequences of **put** and **call** instructions, but we denote a call of a Prolog procedure just by **call proc(Arg**$_1$**, ..., Arg**$_n$**)**, where **Arg**$_i$ is the register or variable holding the argument value.

2.1 Tail Recursion in Regular Prolog Procedures

Tail recursive calls in regular clauses, i.e. those which use environments, are compiled rather inefficiently in the WAM. For example, let us consider a procedure that prints a list to a given stream: Its C code would look as follows:

```
print_list(stream, list)
int                     stream;
register struct pair    *list;
{
    while (list)
    {
        print_item(stream, list->head);
        list = list->tail;
    }
}
```

The Prolog definition is very simple

```
print_list(_, []).
print_list(Stream, [H|List]) :-
    print_item(Stream, H),
    print_list(Stream, List).
```

however its normal translation to the WAM is not satisfactory:

```
print_list/2:
   if ($A_2$ = nil)
      proceed
 * allocate
 * get_variable $Y_1$, $A_1$
   get_list $A_2$
   unify_variable $A_2$
   unify_variable $Y_2$
   call print_item/2
 * put_value $Y_1$, $A_1$
 * put_value $Y_2$, $A_2$
 * deallocate
   execute print_list/2
```

Like in the C version, the procedure uses only two variables, but in Prolog the environment is deallocated after each **print_item/2** call and allocated immediately afterwards. We have marked by an asterisk those instructions, which are unnecessarily repeated on every iteration.

The environment allocated by the recursive subgoal is almost identical to the deallocated one, with exception of $\mathbf{Y}_2$. Clearly, the procedure can be simplified to reuse the same environment and save much of the data traffic.

The first clause can be entered both with and without an environment, and so we either duplicate its code, or allocate even for empty lists, like in C. In this example the first clause is trivial, so we prefer the former.

```
print_list/2:
   if ($A_2$ = nil)
      proceed
   allocate
   get_variable $Y_1$, $A_1$
   get_variable $Y_2$, $A_2$
L1:
   get_list $Y_2$
   unify_variable $A_2$
   unify_variable $Y_2$
```

```
put_value $Y_1$, $A_1$
call print_item/2
if ($Y_2$ = nil) {
    deallocate
    proceed
}
goto L1
```

Note that the variable *Stream* is kept in the environment without change, whereas *List* is destructively changed like in the C version. The control part of the environment, **CE(E)** and **CP(E)** remains unchanged during the whole loop. Thus we can see that if a recursive call reuses its environment,

- it does not have to allocate a new environment, and
- it can keep the loop invariants in the environment.

These two modifications can save considerable amount of data traffic during the loop execution.

3 Recursion Types

In the above example, it was possible to use the fact that an environment in the stack was overwritten with an almost identical environment. Instead of popping and pushing an environment, we could modify the existing one.

Similar optimizations can be performed with other frames in the Prolog machine, provided that a similar situation occurs. In this paper we describe optimizations involving the following frame types:

- environments
- choice points
- delayed environments. A *delayed environment* is a frame used to store the data needed to resume a goal which was suspended.

4 Iteration with Environments

In the above example, we were able to compile the recursive procedure in an efficient way, without any WAM modifications. A necessary condition for this kind of optimizations is that an environment is popped, and another, very similar one is pushed on its place sooner than it is overwritten by another frame. This means that

1. for instance, the last call is directly tail-recursive,
2. the state on deallocation must be deterministic, otherwise the old environment remains in the stack,
3. no environment or choice point must be pushed before the recursively called procedure allocates its environment.

When is the new environment sufficiently "similar"? Obviously, it must have at least the same size, the same control information is guaranteed because it is the last call. Note that even if the call is not recursive, but it calls another procedure with the same environment size, the deallocating and allocating can be optimized away, however this might be easier done using source transformations, e.g. partially eveluating the last call.

Let us consider the third condition more in detail. A choice point can overwrite the deallocated environment in the WAM because both are allocated on the same local stack. The alternative is to use the *split stack model* where the local stack is separated into *environment* and *choicepoint* stack.

Another environment can overwrite the deallocated one only if the last call is not directly recursive, or if the recursive procedure consists of several clauses and the recursive call matches another clause, which also creates an environment. For example (the Browse program by T.Dobry):

```
match([],[]) :- !.
match([X|PRest],[Y|SRest]) :-
        var(Y),!,X = Y, match(PRest,SRest).
match(List,[Y|Rest]) :-
        nonvar(Y),Y = star(X),!, concat(X,SRest,List),
        match(SRest,Rest).
match([X|PRest],[Y|SRest]) :-
        (atom(X) -> X = Y; match(X,Y)),
        match(PRest,SRest).
```

Here at least the third and fourth clause require an environment, and so environment reusing is not straightforward. In such procedures, all clauses requiring an environment have to be merged into one clause. Although this may require a bigger environment, there is no overhead because the environment is allocated only once and only the relevant variables are accessed.

To summarize, here are the main points about environments reusing:

- The tail call has to be recursive. If the recursion is indirect, partial evaluation can be used to convert it into a direct one.
- The state when making the tail recursive call has to be deterministic. Often a cut precedes it, or the left-hand subgoals are only deterministic built-in predicates so that it is straightforward to check this condition. Otherwise, global analysis of the program is necessary. Even procedures which are not deterministic might not be intended to return several solutions, because either their choice point is cut by an ancestor or the program really is nondeterministic, but it should not be so. The global analysis should not only find out if a procedure is deterministic, but it should also check if the choice point of nondeterministic procedures will (or should) ever be used.
- If the procedure contains several regular clauses, they have to be merged into a single clause using disjunctions.
- At the beginning of the loop, the argument registers are saved into the environment. The loop invariants stay there unchanged, the other values are destructively rewritten as soon as the old value is not necessary. At the loop end, the arguments of the recursive call have to be stored into the appropriate cells in the environment rather than in argument registers. On machines with sufficient number of hardware registers the permanent variables should be stored in registers.
- The end condition for the loop is often very simple, most frequently it represents the end of a list or the final value of an index. Then no choice point is necessary for the end loop test. If the procedure is expected to satisfy the end condition frequently enough, or if there is no special action at the loop end, the code for the end clause can be duplicated, once outside of the loop, without allocating an environment, and once inside the loop, where it deallocates the environment and terminates.

5 Iteration with Choice Points

In Prolog, the only way to express a condition is to use alternative clauses, or the if-then-else construct. If the condition is not a simple built-in predicate, a choice point must be pushed before executing it and popped right after. Then, if a tail recursive call follows, we can perform similar optimizations as with the environments. For example (from ILI by S.Haridi):

```
deref(X,E,T) :-  binding(X,E,T1), !,
    deref(T1,E,T).
deref(X,_,X).
```

A code generated according the guidelines in the previous section would be

```
deref/3:
    allocate
    get_variable Y1, A1
    get_variable Y2, A2
    get_variable Y3, A3
L1:
    Y5 := B
    try_me_else L2
    put_variable Y4 A3
    call binding(Y1, Y2, A3)
    cut Y5
    put_value Y4, Y1
    goto L1
L2: trust_me
    get_value Y1, Y3
    deallocate
    proceed
```

As we can see, a choice point is created at the beginning of each cycle, popped after the **binding/3** call and then pushed again in the next cycle with almost identical contents. Let us look at the data contained in this choice point. In the *split stack* model, the choice point contains the following fields:

- **BCP** - code continuation
- **BCE** - environment continuation

○ **A'** - top of the environment stack

○ **BP** - address of the alternative clause

○ **TR'** - trail top pointer

○ **H'** - global stack top pointer

○ Argument registers

In the above example, however, the choice point can store less information, and instead of pushing and popping we can just update its contents, exactly as we do with the environments. **BCP** is identical with **CP(E)**, so it is redundant, the arguments are also contained in the environment. Therefore, only **BCE**, **A'**, **TR'**, **H'** and **BP** have to be stored in the choice point, and only **TR'**, **H'** might have to be updated after each cycle. The optimized code then looks as follows:

```
deref/3:
    allocate
    get_variable $Y_1$, $A_1$
    get_variable $Y_2$, $A_2$
    get_variable $Y_3$, $A_3$
    Partial_ChP(E, A, L2)
    $Y_5$ := B
L1:
    Update_ChP(TR, H)
    put_variable $Y_4$ $A_3$
    call binding($Y_1$, $Y_2$, $A_3$)
    cut $Y_5$
    put_value $Y_4$, $Y_1$
    goto L1
L2: trust_me
    get_value $Y_1$, $Y_3$
    deallocate
    proceed
```

If **binding/3** is known not to bind any variables and not to push any structures on the global stack, **TR'** and **H'** can be omitted in the choice point and **Update_ChP** is void. If it is known to be deterministic, **A'** can be omitted and the cut is also void.

In this way, a series of choice points can be collapsed into one. This is possible only under the following conditions:

- When the recursive call is made, there must be no choice point left, so that the state is as when entering the procedure for the first time. This also means that the subgoals must either be deterministic, or there must be a cut before the tail recursive call.
- There must be no failure between popping the choice point and pushing a new one. The choice point can be popped either by the implicit cut in in if-then-else, by a normal cut, or in the last clause.

To optimize choice point handling in procedures like

```
loop :- (end -> stop; process, loop),
```

process/0 must be deterministic and must be known to always succeed, since when it fails, the whole procedure must fail but we still keep a choice point for **loop/0**.

A slightly different situation can occur with simple predicates like **member/2**:

```
member(X, [X|_]).
member(X, [_|T]) :-
    member(X, T).
```

If **member/2** is used to generate values, the choice point can obviously be reused, because after popping it and making a recursive call, an almost identical choice point is pushed again. When the second argument is not a list, **member/2** must fail, however when the choice point is reused, it is always there and so it has to be popped explicitly.

The recursive choice point is identical to the previous one except for the value of $\mathbf{A}_2$. Although it would be possible to make a destructive change in the choice point itself, a more suitable method is again to use an environment to hold the arguments:

```
member/2:
    allocate
    get_variable Y1, A1
    get_variable Y2, A2
    try_me_else L3
L1:
    get_list Y2
    retry_me_else L2
    unify_value Y1
    unify_variable Y2
    proceed
```

```
L2: retry L1
L3: trust_me
    fail
```

Note how **BP(B)** is handled to make the whole predicate fail if the second argument is not a list.

We have measured the frequency of reusable environments and choice points on a set of small programs from similar work by Touati and Despain [4] and on a set of middle-to-large programs. We list the total number of environments *Env* and choice points *ChP* and the percentage of reusable ones.

Small Program	Env	ChP	Reusable Env	Reusable ChP
concat	2	5	0.0 %	80.0 %
differen	23	62	0.0 %	19.4 %
hanoi	509	255	0.0 %	24.9 %
mumath	203	562	0.0 %	69.0 %
nrev1	30	0	0.0 %	0.0 %
palin25	105	76	72.6 %	80.3 %
primes	362	842	73.0 %	57.1 %
qsort	50	229	51.0 %	86.0 %
queens	64	201	21.5 %	3.0 %
query	27	676	0.0 %	0.0 %
Program	**Env**	**ChP**	**Reusable Env**	**Reusable ChP**
Boyer	132517	138628	42.0 %	0.3[1] %
Browse	210919	251899	35.4 %	57.1 %
General	13075	24033	(26.8[5] %) 9.7 %	(63.3[5] %) 12.0 %
TT	27140	14710	0.0[2] %	16.5 %
ILI	4145	2949	5.2[3] %	8.1[3] %
Edf	5857	29040	(22.5[5] %) 9.1 %	74.8 %
Spreadsheet	42655	13895	23.3 %	23.0 %
Plm Compiler	92401	164619	18.2 %	49.4 %
Pdbs	29196	82897	57.3 %	74.7 %
Mvv	96443	6596	0.2[4] %	36.1 %
Toesp	375221	194836	17.8 %	16.3 %
SB Prolog	138883	255929	20.0 %	55.0 %
Tp	114027	128479	80.2 %	64.6 %
Chat	43821	42877	22.9 %	21.1 %
Theorem Prover	50036	61122	22.7 %	39.3 %
Sicstus	15546	17391	26.0 %	30.3 %
Aunt	123483	176054	55.1 %	49.9 %
Simulator	20209	35794	32.9 %	20.1 %

We can see among others, that measuring small programs can differ significantly from large ones. The ratio of reusable choice points is sufficiently high, for environments it is sometimes necessary to perform program transformations to achieve maximal effect.

Our figures for environments differ from Touati ones, both on identical programs and on the average of reusable environments in larger programs. The reasons for this, as far as we can tell, are

- Touati assumes that an environment is allocated for every clause with at least two subgoals. We, on the other hand, assume that most of the built-in predicates can be handled differently, as is the case in SEPIA [2], and so we allocate less environments.
- Touati assumes that failure in the body of the tail-recursive goal prevents environment reusing, and so e.g. in

  ```
  loop(X) :- end(X), !.
  loop(X) :- process(X, X1), loop(X1).
  ```

  the environment is not counted as reusable. However, if we reuse the choice point, the environment can be reused as well.

  Due to the same reason, Touati does not count procedures with several regular clauses, but we do count them.

[1]Almost all of the involved choice points are created only by an inefficient compilation, with improved clause selection there would be only about 500 choice points.

[2]This program is highly nondeterministic and it includes indirect recursion and metacalls. When it is partially evaluated, it uses only a fraction of the environments and choice points.

[3]This program uses heavily a procedure that applies a substitution on a term. Since the term arguments are converted to lists using **=../2**, no actual tail recursion takes place. This could be improved by program transformation.

[4]This program simulates parallel execution by its own version of the **bagof/3** predicate which uses heavily the metacall, and this always allocates an environment.

[5]The numbers in parentheses result from cosmetic changes in the original programs, e.g. removing an obsolete cut or replacing assert/retract by a counter.

6 Iteration with Suspended Goals

Prolog systems that support goal suspension can also benefit from efficient loops. When a goal delays, some data has to be pushed on one of the stacks, which enables the system to resume the goal later. This data, called in SEPIA a *delayed environment*, must always contain at least the goal arguments and the procedure identification. When a delayed goal is woken by binding of a variable, its arguments are fetched from the delayed environment and the goal is resumed. If there is no choice point younger than its delayed environment, the delayed environment is no longer accessible and it becomes garbage (because it might not be at the stack top and so it cannot be popped). However, if the resumed goal makes a recursive call which itself delays, instead of pushing a new delayed environment it is possible to reuse the old one, which not only decreases the amount of garbage, but it also saves memory accesses because the procedure identification is the same and so can be some of the arguments. Here is a simple example:

```
int_list(N, [N|L]) :-
    N1 is N + 1, int_list(N1, L).
primes :-
    sieve(L), int_list(2, L).
delay sieve(L) if var(L).
sieve([X|L]) :-
    write(X), write(' '),
    filter(X, L, L1), sieve(L1).
delay filter(_, Li, _) if var(Li).
filter(P,[N|LI], LL) :-
    (N mod P =\= 0 ->
        LL = [N|NLI], filter(P,LI, NLI)
    ;
        filter(P,LI, LL)
    ).
```

This program generates a list of all prime numbers. Each call to **sieve/1** removes from the lazily generated list all integers divisible by a new prime. Both **sieve/1** and **filter/3** are suspended when they are first called. When the corresponding variable is instantiated, they are woken, call their body subgoals and finally a tail recursive call is made which again delays. After waking each goal, its delayed environment is not accessible and can be reused when delaying the recursive call. We have measured with the SEPIA system the gain of this modification when generating all primes less than 10000:

	Copying	Reusing	Gain
Number of GC	193	55	72 %
Total amount of garbage	96620016	27556424	72 %
Time spent in GC	36.8s	12.8s	65 %
Total elapsed time	187.7s	170.0s	10 %

In programs that use coroutining, tail-recursive procedures that always delay is a fairly common case. If the delayed environment can be reused, the program can benefit from destructive assignment at the implementation level, while being completely declarative at the source level.

7 Conclusion and Future Work

We have described in this paper how repetitive actions in Prolog can be compiled efficiently, so that the generated code is close to that of imperative languages. Since the only way to express such actions in Prolog is recursion, the Prolog compiler has to be particularly careful when compiling recursive procedures, because most often Prolog programs spend a considerable amount of time in them.

Recent work of Millroth [3] shares several features with our work. It is more complex because it expands the recursion into parallel iteration, it considers nonlinear recursion as well and he presents an actual algorithm for the compilation. On the other hand, its approach is simpler because it considers only loops with environments, where the iteration is made only by decrementing an integer variable.

To achieve a satisfactory compilation of recursive procedures, there is still a lot of work to be done. This includes mainly:

- Classification of possible recursion types, possibly mixing environment, choice points and suspended goals and designing code patterns to be generated for them.
- Analysing existing Prolog programs and designing program transformations that are necessary for the compiler to compile loops efficiently. Often a simple transformation, like adding or removing a cut, can considerably speed up the loop execution.

  The number of environments can be often drastically reduced by unfolding non-recursive procedures. Program transformations similar to that one in [1] which convert non-tail recursive loops into tail recursive ones prove extremely useful.
- Specifying what information the compiler needs to compile the loops. Some of this information can be local to a procedure, some can be provided only by global flow analysis.
- Change the way backtracking is compiled. As we have seen in the examples, some of the choice point accesses can be avoided, resulting in faster code, however the WAM is not fine-grained enough for this purpose.
- Design the actual algorithms to compile recursive procedures and implement them.

Acknowledgements

The author would like to thank Joachim Schimpf for valuable comments in many discussions about the WAM and Prolog and Alexander Herold for thorough reading and commenting a previous version of the paper.

Bibliography

[1] Saumya K. Debray. Unfold/fold transormations and loop optimizations of logic programs. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 297–307, Atlanta, June 1988.

[2] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, San Francisco, August 1989.

[3] Håkan Millroth. *Reforming Compilation of Logic Programs.* Uppsala Theses in Computer Science. UPMAIL, 1990. Ph.D. thesis.

[4] Hervé Touati and Alvin Despain. An empirical study of the Warren Abstract Machine. In *Proceedings 1987 Symposium on Logic Programming*, pages 114–124, San Francisco, September 1987.

[5] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI, October 1983.