On Compiling Indexing and Cut for the WAM

Mats Carlsson

SICS, Swedish Institute of Computer Science
PO Box 1263
S-16313 Spånga, Sweden

Phone: +46 - 8 - 7507970
E-mail: mats-c@sics.uucp

Abstract

There exist several proposals for the treatment of clause indexing in a compiler for the WAM. These are discussed, and a particular proposal is advocated, which postpones as long as possible the creation of choicepoints. We then address some problems with implementing the cut operator. Finally, it is shown how indexing instructions can subsume certain body goals.

Keywords: logic programming, abstract machines, compilers, Prolog.

§1 Introduction

We are considering the "New Engine", or WAM, for Prolog by D.H.D. Warren. In [Warren 83], an abstract machine is defined consisting of a memory model and about 40 instructions. We assume herein that the reader is familiar with the WAM. A very good exposition of the WAM is given in [Gabriel *et.al.* 84].

Warren's paper contains an account of clause indexing. Slightly different approaches have been taken by other groups. We will compare current approaches and present herein yet another one, and show certain compiler optimizations that are possible.

Cut is not treated in [Warren 83] but has been subsequently added by other groups in slightly different ways. We will compare these and present our own. Finally, it is shown how cut under some circumstances can be compiled into indexing instructions.

1.1 Preliminaries

We assume familiarity with standard Prolog implementation terminology. In particular, we mean by the *type* of a term its meta-logical status as variable, constant, list, or structure. By the *principal functor* of a non-variable term T we mean the name and arity of the principal function symbol of T. A computation of a predicate is said to be *determinate* if no backtracking alternatives for the computation exist.

We use the following abbreviations for the principal WAM registers:

A1, ***A2***, ..	argument registers
P	program pointer
CP	continuation program pointer
E	current environment
B	current choicepoint
TR	top of trail
H	top of heap

and denote the corresponding choicepoint fields by ***Ai'***, ***P'***, etc.

1.2 Proposals for clause indexing

An important part of Prolog implementations is clause indexing, i.e. to filter out, for given arguments, the set of clauses that potentially match those arguments. In WAM, as in most implementations of Prolog, only the principal functor of the first argument is used as index key. Indexing on multiple arguments has been proposed by several authors, and would arguably be of great value in several applications, but is not treated in this paper.

1.2.a The Warren Method

Warren's indexing instructions consist of six instructions for managing choicepoints and three instructions for discriminating on the first argument. We give here a brief synopsis of his indexing instructions:

`try_me_else L`
: Establish a choicepoint where the alternative program pointer $\boldsymbol{P'} = L$.

`retry_me_else L`
: Set $\boldsymbol{P'} = L$ in the current choicepoint.

`trust_me_else_fail`
: Remove the current choicepoint.

`try L`
: Establish a choicepoint where the alternative program pointer $\boldsymbol{P'}$ points at the next instruction. Proceed at *L*.

`retry L`
: Set $\boldsymbol{P'}$ to point at the next instruction. Proceed at *L*.

`trust L`
: Remove the current choicepoint. Proceed at *L*.

`switch_on_term Lv,Lc,Ll,Ls`
: ***A1*** is dereferenced, and a branch is taken depending on its type as variable (`Lv`), constant (`Lc`), list (`Ll`), or structure (`Ls`).

`switch_on_constant N,[C1: L1, C2: L2,... Cn: Ln]`
: If there is an i such that $\boldsymbol{A1} = Ci$, a branch to Li is taken, otherwise the engine backtracks. ***A1*** has already been dereferenced to a constant.

`switch_on_structure N,[C1: L1, C2: L2,... Cn: Ln]`
: If there is an i such that the principal functor of $\boldsymbol{A1} = Ci$, a branch to Li is taken, otherwise the engine backtracks. ***A1*** has already been dereferenced to a structure.

The indexing scheme is best explained by quoting from [Gabriel *et.al.* 86]:

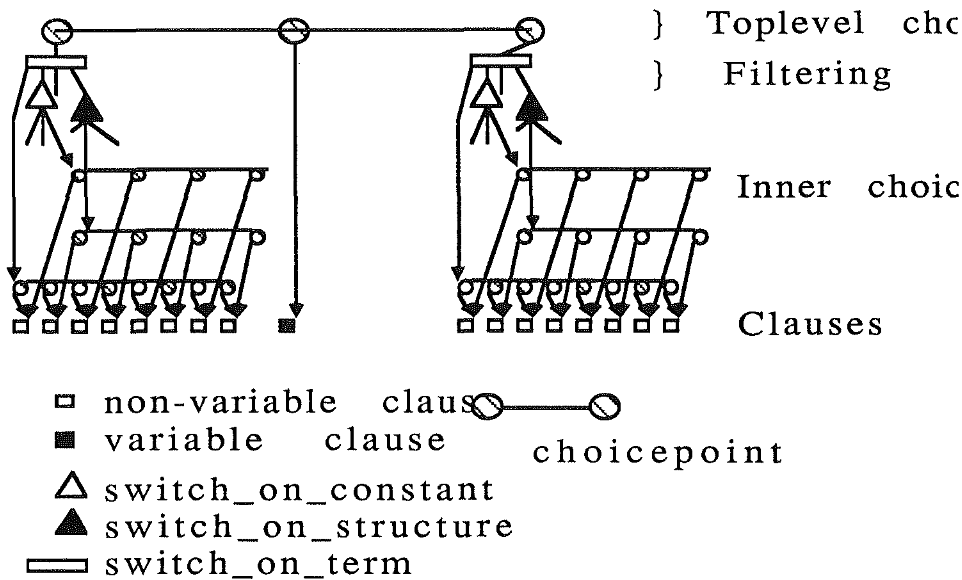
"Suppose that the clauses in a given procedure are $C_1, C_2, \ldots C_n$. These are broken into groups $G_1, G_2, \ldots G_m$. Each group is either a single clause with a variable occurring as the first argument of the head literal, or a set of clauses in which none of the clauses contains a variable as the first argument of the head literal. These groups result in the following generated code:

```
        try_me_else         L2
        <code for G1>

L2      retry_me_else       L3
        <code for G2>
        :
        :
Lm      trust_me_else fail
```

```
        <code for Gm>         "
```

The following picture displays a situation with a single clause with a variable as first head argument in preceded and followed by other clauses. The variable clause introduces an alternative in the toplevel choicepoint.



If there is just one group, the above scheme is simplified, since there is no need to establish a choicepoint. Groups that are not single clauses compile to code for filtering out using *switch* instructions the set of clauses that can possibly match a given first argument. For such sets that are not singletons, a choicepoint is established using *try - retry - trust* instructions such that the engine may backtrack over the possible matches.

However, if this choicepoint belongs to a group other than the last one, the choicepoint will be redundant except for the ***P'*** field, since the `try_me_else` instruction already established a choicepoint. This is an undesirable phenomenon, particularly since it has been shown that choicepoint management is expensive compared to other operations of the WAM [Tick 86]. The redundancy has been recognized and an approach for storing only the non-redundant information is described in [Turk 86].

1.2.b The Berkeley Method

In Van Roy's compiler [Van Roy 84], clauses are arranged into groups with the additional constraint that no two clauses in a group may have first arguments with the same principal functor. Thus he avoids ever creating two choicepoints before entering a clause, at the price of sometimes having more backtracking before a matching set of clauses, than in the Warren method. Thus the price of avoiding redundant choicepoints is a degradation of the overall efficiency of clause

indexing due to the extra constraint.

1.2.c The Syracuse Method

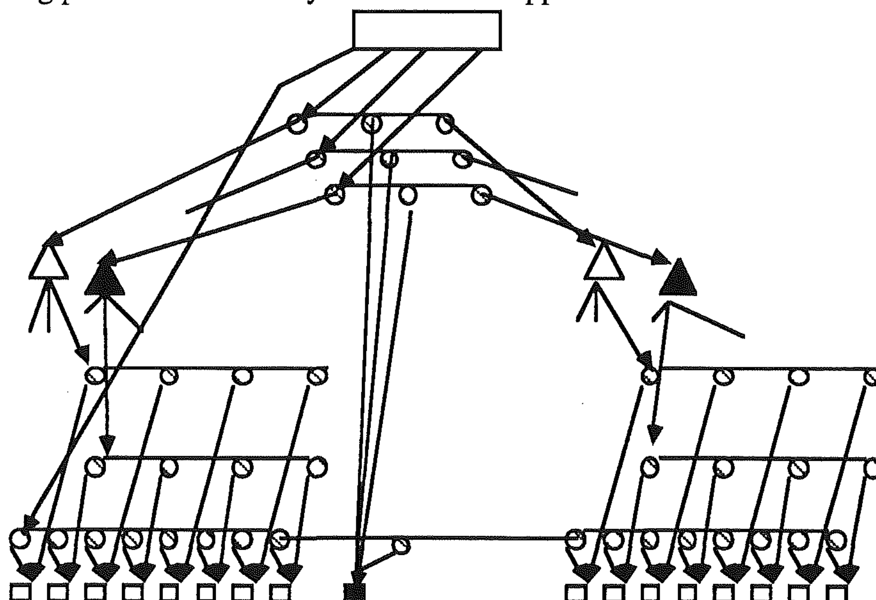
In [Bowen *et.al.* 86], the same indexing instructions are used as in the Warren method. A similar two-level indexing with the same kind of redundancy is used, however the clause group partitioning is governed by the type of the first argument, rather than the principal functor. The indexing scheme discriminates on the type of the first argument, and only creates an outer choicepoint if more than one group can match. For example, a predicate with five clause groups

```
G1:  constant group
G2:  structure group
G3:  variable group
G4:  constant group
G5:  list group
```

would compile to:

```
        switch_on_term Lc1,Lk,Ll,Ls
Lk      try Lg1                % outer choice
        retry Lg3
        trust Lg4
Ll      try Lg3                % outer choice
        trust Lg5
Ls      try Lg2                % outer choice
        trust Lg3
Lg1     switch_on_constant ...
Lg2     switch_on_structure ...
Lg3     try ...                % inner choice
Lg4     switch_on_constant ...
Lg5     try ...                % outer choice immediately after a trust
Lc1     try_me_else Lc2
        <code for first clause>
        :
        :
```

The following picture shows the Syracuse method applied to the same situation as before:



Thus the creation of outer choicepoints is postponed after the initial `switch_on_term` instruction. The price for this is slightly increased code size, since the outer try sequence has been "distributed out" to the different cases of type of the first argument. By taking this process one step further, two-level indexing can be avoided and the redundancy problem eliminated. This leads us to our proposed method.

1.2.d One-Level Indexing

By discriminating first on the type of the first argument, and second, when appropriate, on its principal functor, one can filter out the set of potentially matching clauses. A choicepoint is then needed only for non-singleton sets.

The instruction set needs a slight change to implement this scheme, since a principal functor which is not in the set of principal functors occurring in the source code as first head argument still matches clauses with a variable as first head argument. Thus, we use the modified instructions:

```
switch_on_constant N,Table,DefaultLabel
switch_on_structure N,Table,DefaultLabel
```

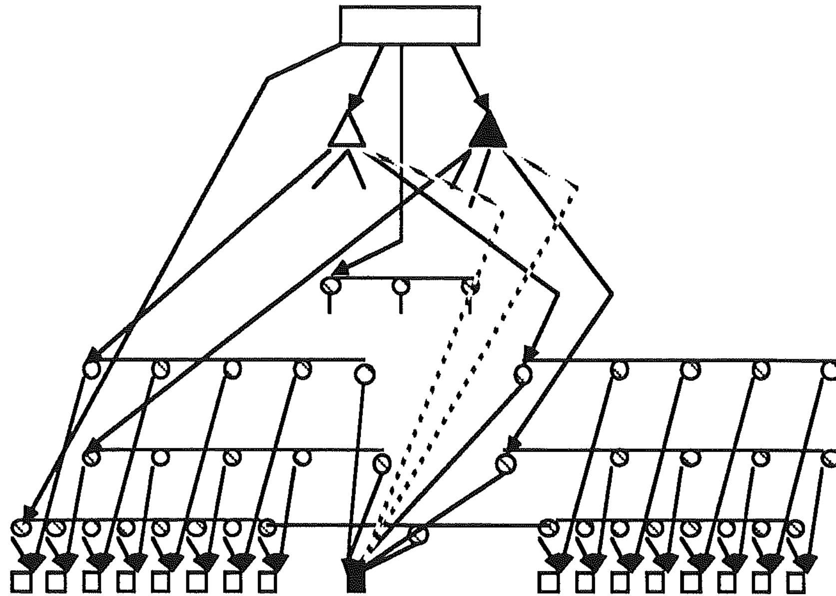
in clause indexing, where `DefaultLabel` typically corresponds to all the clauses with a variable as first head argument, i.e. a `try` sequence, a single clause, or `'fail'`, if there is no such clause. For example, compiling

```
p(a) :- ...
p(X) :- ...
p(b) :- ...
```

yields

```
        switch_on_term Lc1,Lk,Lc2a,Lc2a
Lk:     switch_on_constant 2,[a: Lk1 ,b: Lk2],Lc2a
Lk1:    try Lc1a
        trust Lc2a
Lk2:    try Lc2a
        trust Lc3a
Lc1:    try_me_else Lc2a
Lc1a:   <code for clause 1>
Lc2:    retry_me_else Lc3a
Lc2a:   <code for clause 2>
Lc3:    trust_me_else_fail
Lc3a:   <code for clause 3>
```

The following picture shows our proposed method applied to the same situation as before. Dotted lines denote `DefaultLabel` references:



Again, the price is an increase in code size, since the number of indexing instructions is generally $O(n^2)$, where n is the number of clauses. There is a quadratic factor since the clauses referred to by `DefaultLabel` have to be included in every other clause set. However, the worst case situations seem quite rare, and it is our experience that the number of indexing instructions is normally "almost linear" in the number of clauses. The compiler could detect when the quadratic factor becomes a problem and then choose one of the other methods.

1.3 Proposals for Cut

Another important part of Prolog implementations is the cut operation, the operational semantics of which is yet to be laid down firmly [Moss 86]. Informally, the cut operation renders the predicate in which it occurs determinate. In terms of the memory model of the WAM, the cut excises backtracking information from the local stack and from the trail.

1.3.a Berkeley Cut

The Berkeley implementation [Van Roy 84] has chosen a different operational semantics for cuts inside disjunctions: such a cut commits the innermost disjunction in which it occurs, rather than committing the entire predicate in which it occurs. See [O'Keefe 85] for an enlightening discussion on this issue.

For cuts outside disjunctions, the compiler ensures the presence of an environment containing a saved value of ***B*** and compiles *cut* into:

```
cut
```

The machine has a state bit telling whether the choicepoint pointed to by the environment should be

deleted by the instruction or not. This bit is manipulated by several of the indexing and procedural instructions.

For cuts inside disjunctions, the compiler knows where the machine would backtrack to, and compiles *cut* into:

```
cut BacktrackLabel
```

This instruction traverses the chain of choicepoints until one is found whose ***P'*** slot matches `BacktrackLabel`. This method however constrains each disjunct to contain only one *cut* and constrains the last disjunct to not contain any *cut* at all.

1.3.b Syracuse Cut

The Syracuse implementation [Bowen *et.al.* 86] has added a new machine register, *cutpt* , and instructions to manipulate it. The *cutpt* register is saved in every choicepoint and restored upon backtracking. The call and execute instructions copy the value of ***B*** into *cutpt*. .

Now, *cutpt* contains the address of the last choicepoint created before entering a procedure. The cut operation is then done by copying this value back to the choicepoint register, having saved it in the environment first if the cut was preceded by a goal. Special-purpose instructions take care of the different cases. The trail is not mentioned in conjunction with these instructions, but we take for granted that it is tidied as well.

We have two observations to make. First, the architecture has been extended by a new register, a new choicepoint field, and extra work in `call` and `execute`. The extensions cause some extra work even for predicates that do not use cut. Furthermore, the scheme requires all inferences to pass a `call` or `execute` instruction, ruling out the possibility of an indexing instruction referring to a predicate instead of a local label. Second, in the WAM memory model with the local stack containing both environments and choicepoints, the cut operation sometimes renders some of these inaccessible when they are not at the top of the stack. For example, in the predicate

```
p(X,Z) :- p1(X), !, q1(X,Y), r(Y,Z).
p(X,Z) :- q2(X,Y), r(Y,Z).
```

the first clause will create an environment which prevents the choicepoint storage from being reclaimed even though the cut makes it inaccessible. It will be reclaimed at the tail-recursive call to `r/2` only if `q1/2` succeeds determinately. The situation becomes worse for cuts inside disjunctions. Pathological examples can be made up where stack overflows are caused by backtracking information which is inaccessible after a cut but cannot be reclaimed.

1.3.c Belgian Cut

With [Debray & Warren 86] and [Barklund & Millroth 86], we take an approach similar to the Syracuse method, with two changes. First, instead of copying at every inference the current value of $\boldsymbol{B}$ into a new register, we use the following scheme: An instruction

```
choice An:   An := B;
```

where n is the number of the first free argument register, begins the code for predicates using cut. The rest of the code is as if the predicate had an extra argument to be used in a subsequent instruction

```
cut Vn:  B := Vn; tidy trail;
```

This method is due to Venken [Venken 84] where the operations are called `mark(L)` and `!(L)`, therefore we call it Belgian Cut.

Thus, the extra work incurred by the Syracuse method on predicates not using cut is avoided by separating out the `choice` operation and doing it only in predicates that use cut. Indexing instructions may now refer directly to predicates also.

Our second change addresses the problem of reclaiming stack space. This problem can be partially solved by splitting the local stack into an *environment stack* and a *choicepoint stack* . This rather drastic change in the memory model calls for a slight change in the choicepoint format: The $\boldsymbol{B'}$ slot can now be omitted, since it can be deduced from $\boldsymbol{B}$ and the arity of the choicepoint. However, to compute $\boldsymbol{A}$, i.e. the effective top of the environment stack, it is no longer possible to compare $\boldsymbol{B}$ and $\boldsymbol{E}$. Instead, we compute $\boldsymbol{A}$ by the formula

$$\boldsymbol{A} = max(\boldsymbol{A'}, \boldsymbol{E} + env_size(\boldsymbol{CP}))$$

where $\boldsymbol{A'}$ is a choicepoint slot containing the effective top of stack at the time the choicepoint was created. Similarly the trailing condition for a variable X that dereferences to the stack becomes

$$X < \boldsymbol{A'}$$

Since this test now involves a memory reference, it may be worth while to have a shadow register

for A'. Warren suggests a shadow register for H' for the same reason.

With split stacks, the cut operation always deletes a number of the most recent choicepoints i.e. from the top of the choicepoint stack, and so the storage is immediately reclaimed. However, in the case of cut inside disjunctions, there may still be some space on the environment stack that is not immediately reclaimed. For example, assume in

```
p(X) :- q(X), (r(X), !, s(X), s1(X); t(X)), u(X).
p(X) :- v(X).
```

which our compiler transforms to

```
p(X) :- $choice(Y), p'(X,Y).

p'(X,Y) :- q(X), p"(X,Y), u(X).
p'(X,_) :- v(X).

p"(X,Y) :- r(X), $cut(Y), s(X), s1(X).
p"(X,_) :- t(X).
```

that `q/1` and `r/1` succeed non-determinately, i.e. that they have left choicepoints, environments, and trail information when we arrive at the cut operation. The cut deletes all of these, except the environments possibly left by `q/1`, since those environments are protected by the environment created by the first clause of `p"/2`. The space is eventually reclaimed at the tail-recursive call to `s1/1`, provided that `s/1` succeeds determinately.

To reclaim the space immediately after cut would involve a more complicated "remote cut" instruction that would move environments, and such an operation would probably be too costly.

Alternatively, disjunctions could be compiled more cleverly as suggested in [Van Roy 84] to avoid the extra environment, although that would seem to add significant complexity to the compiler.

The implementation of [Barklund & Millroth 86] maintains A as a register, instead of computing it when needed. Two variants of the `cut` instruction are proposed: plain `cut`, used when a subsequent `call` or `deallocate` will update the A register, and a variant which combines the `cut` and `trim` operations, used before `proceed` or `execute`. This seems to be an attempt at reclaiming inaccessible environments, but would not help in the `p/1` example above. Barklund and Millroth also devise a method for reclaiming heap storage in conjunction with cut.

Finally, we note that if-then-else constructs never run into these problems since the cut is always "local" there, and agree with O'Keefe [O'Keefe 85] that constructs with cuts inside disjunctions represent doubtful programming style.

§2 Compiler Optimizations

There is a host of possible compiler optimizations involving indexing instructions.

2.1 Indexing Subsuming Type Tests

Meta-logical type tests on the first head argument can typically be incorporated into clause indexing. For example,

```
p(X,Y) :- var(X), q(X,Y).
p(X,Y) :- nonvar(X), r(X,Y).
```

is compiled by our compiler to

```
p/2:    switch_on_term q/2,r/2,r/2,r/2
```

This optimization is only done if the type test is the first goal, to cater for goals with side-effects. Note that the `var(X)` and `nonvar(X)` test must be done at run-time if the head arguments are not all distinct variables, to cater for cases like:

```
:- q(X,1).

q(X,X) :- var(X), v(X).
q(X,X) :- nonvar(X), w(X).
```

Other authors have proposed to generalize the `switch_on_term` instruction to any temporary or permanent variable, thus coding in-line all meta-logical type tests. Some implementations have made the instruction more fine-grained by separating between more basic types.

2.2 Indexing Subsuming Cuts

Sometimes the cut operation need not be translated into special instructions, but can be incorporated into clause indexing instead. For example,

```
p(X,Y) :- var(X), !, q(X,Y).
p(1,Y) :- !, p1(1,Y).
p(2,Y) :- p2(2,Y).
p(X,Y) :- r(X,Y).
```

compiles to

```
p/2:    switch_on_term q/2,Lc,r/2,r/2
        switch_on_constant 2,[1: p/2, 2: Lc1],r/2
Lc1:    try p2/2,2
```

```
trust r/2,2
```

2.3 Overloaded Permanent Variables

Permanent variables with disjoint lifespans could be allocated to the same environment slots if the code is determinate. Cut could act as a fence between lifespans in such cases. This optimization has not been added to our compiler, as the potential stack space savings seem rather marginal.

2.4 Specialized Get Instructions

The clause code can capitalize on the fact that clause indexing is performed. When a clause whose first head argument is a list, say, is entered, clause indexing guarantees that A_1 is dereferenced and that its type is either variable or list. Whereas the clause code would normally start with `get_list A1`, we use the specialized `get_list_a1` instruction which represents a much simpler operation than the general instruction. Our native code generator takes further advantage of the specialized instruction and generates different code streams for the two cases (variable or list). The other general *get* instructions (except `get_variable` and `get_value`) have similar counterparts.

§3 Concluding Remarks

We have discussed some approaches for clause indexing and have advocated one which postpones as long as possible the creation of choicepoints. We discussed implementations of cut and in particular addressed the problem of reclaiming stack storage.

An issue not treated by Warren is the *arity* of choicepoints. Several implementations add the arity as an extra argument of `try(_me_else)` and store it in the choicepoint. An alternative which uses less dynamic space but more static space is to add an arity argument to `retry(_me_else)` and `trust(_me_else)` also. The arity argument can then be accessed as an offset from P'. We advocate this second approach.

A final observation on the instruction set: The instructions `try_me_else`, `retry_me_else`, and `trust_me_else` are not strictly necessary since they can be replaced by `try/retry/trust`, but will optimize performance on pipelined hardware.

Acknowledgements

The author is indebted to Jonas Barklund, Lee Naish, Ross Overbeek, David Warren, and to his colleagues at SICS for discussing and refining the ideas reported herein.

References

[Barklund & Millroth 86] J. Barklund, H. Millroth, *Garbage Cut for Garbage Collection of Iterative Logic Programs* , Proc. IEEE Symposium on Logic Programming, Salt Lake City 1986.

[Bowen *et.al.* 86] K.A. Bowen, K.A. Buettner, I. Cicekli, A.K. Turk, *The design of a high-speed incremental portable Prolog compiler* , Proc. 3ICLP, London, 1986.

[Debray & Warren 86] S.K. Debray, D.S. Warren, *Detection and Optimization of Functional Computations in Prolog* , Proc. 3ICLP, London, 1986.

[Gabriel *et.al.* 84] J. Gabriel, T. Lindholm, E.L. Lusk, R.A. Overbeek, *Tutorial on the Warren Abstract Machine for Computational Logic* , ANL-84-84, Argonne National Laboratory, Argonne IL, 1984.

[Moss 86] C. Moss, *CUT & PASTE - defining the impure primitives of Prolog,* Proc. 3ICLP, London, 1986.

[O'Keefe 85] R.A. O'Keefe, *On the treatment of cuts in Prolog source-level tools,* Proc. IEEE Symposium on Logic Programming, Boston, 1985.

[Tick 86] E. Tick, *Memory performance of Lisp and Prolog programs* , Proc. 3ICLP, London, 1986.

[Turk 86] A.K. Turk, *Compiler Optimizations for the WAM* , Proc. 3ICLP, London, 1986.

[Van Roy 84] P. Van Roy, *A Prolog Compiler for the PLM* , M.Sc. Thesis, Dept. of Computer Science, University of California, 1984.

[Venken 84] R. Venken, *A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source-to-Source Transformation and Qurey Optimization* , Proc. ECAI 84.

[Warren 83] D.H.D. Warren, *An Abstract Prolog Instruction Set* , SRI International #309, 1983.