Upmail Technical Report no. 35

Compilation for Tricia and its Abstract Machine

Mats Carlsson[1]

UPMAIL
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computing Science, Uppsala University
P.O. Box 2059
S-750 02 Uppsala, Sweden

[1]Author's present address:
Swedish Institute of Computer Science
P.O. Box 1263
S-163 13 Spånga, Sweden

Table of Contents

Abstract

An elaboration of the Warren Abstract Machine for Prolog is described along with its compiler. The elaborations include arithmetic, cut and delay operators, and constraints imposed by a garbage collector. Various compiler optimizations have been introduced and are discussed.

1. Introduction

This paper describes the abstract machine and front-end compiler of a compiler-based Prolog implementation effort nicknamed Tricia. The motivation behind the effort is

1. To develop expertise in compiler technology for Prolog.
2. To build a Prolog system with high performance and high functionality to serve as a basic software tool.
3. To explore whether new control structures such as coroutines can be included without performance penalty in standard Prolog implementations.

The abstract machine is a derivative of the Warren Abstract Machine [War83]. It elaborates issues not treated in [War83], such as arithmetic, the cut operator, and constraints imposed by a garbage collector. We have also introduced delay primitives [Nai85] in the implementation, which includes a compiler, a back-end for generating Digital PDP10 machine code (KL10 processor with Extended Addressing) and a runtime system for it.

2. Abstract Machine Architecture

The architecture is substantially the same as in [War83]. This chapter describes the main registers, main data areas, main data structures, and memory word format. It corresponds to Chapters 2, 3, and 4 in [War83]. The level of detail is chosen to suit the description of the various instructions.

2.1 Data objects

A Prolog object is represented by a word containing a *tag* and a *pointer*. The pointer is usually a memory address, however for some datatypes the pointer contains data.

2.1.1 Logical Variables

The following three tags are used in the representation of logical variables:

$var
This represents a heap variable. The pointer points to a heap word

<value>

where `value` is a self-reference if uninstantiated.

$sva
This represents a stack variable. The pointer points to a stack word

<value>

where `value` is a self-reference if uninstantiated.

$clo
This represents a goal which is suspended on a variable, aka a *constrained variable*. The pointer points to a heap vector

<value, $lst(prev), goal>

where `value` is a self-reference if uninstantiated, `goal` is the suspended goal, and `prev` is a reference to the most recent `$clo` object.

2.1.2 Terms

The following tags are used for representing terms in programs, i.e. objects that have a syntax in the language.

$chr
A character object, including font information and meta bits.

$int
An integer (actually small integers have a special tag but this is invisible).

$nil
The empty list.

where CE is the continuation environment, CL is the continuation program address, Yi are variables that need to be saved over procedure calls.
NB. At procedure calls, the size of the vector is accessible at an offset from the machine word referred to by L. This size is decremented for each procedure call by Warren's "trimming" mechanism.

A *choice point* is a stack vector

<P', A', TR', H', E', L', X0', ..., Xm'>

where Xj' are saved argument registers, and the other fields correspond to previous values of machine registers.

2.3 Registers

P
program pointer (to the code area)

L
continuation program pointer (to the code area)

E
last environment (on the environment stack). Actually points to last word before it. $sva is currently zero, which is optimal for creating variables on the environment stack.

B
last choice point (on the choicepoint stack). Points to first word of it.

A
Top of stack.

TR
Top of trail.

H
Top of heap.

S
Structure pointer (to the heap).

X0..Xn
Argument/temporary registers

F
General event register. One of the bits is on if there are goals to wake up. One of the bits is on if an abort has been signalled. One of the bits is on if the heap need to be garbage collected. Etc.

W
Holds a goal to be woken iff the corresponding F flag is raised. Resides in memory.

$lst
A non-empty list. The pointer points to a vector
<car, cdr>
containing head car and tail cdr of the list.

$sym
This represents a symbolic constant. It is considered atomic for the purposes of this paper. Symbols are *unique*, i.e. two symbols have the same machine address iff they have the same names.

$str
A structure, i.e. an n-ary function (n > 0). The pointer points to a vector
<functor, arg.1, ..., arg.n>

$sst
A string, i.e. a compact list of ascii characters. It is considered atomic for the purposes of this paper. Strings are not unique.

The tag **$oth** is used for special terms of some kind (array, hash table, or the like), which do not have syntax.

2.2 Data Areas

The different data areas are laid out in memory as follows:
<code area, heap, environment stack, choicepoint stack, trail>.
The various areas are used as follows:

code area
Contains compiled code and structure-shared constants and pure data. Should be garbage collectable.

heap
Contains all data created by unification. It shrinks upon backtracking and is garbage collected when full.

environment stack
Contains environments (continuation + vector of value cells). Also used as PDL in unification etc. There is no garbage collection in this area.

choicepoint stack
Contains choice points (saved machine states). There is no garbage collection in this area.

trail
Contains pointers to variables that we might unbind. Other side-effects are occasionally trailed too. Things to do "upon backtracking" are put here too.

An *environment* is a stack vector
<CE, CL, Y0, ..., Yn>

3. Tricia Instructions and Basic Operations

This chapter corresponds to Chapter 8 in [War83].

The semantics of each instruction is given in a pseudo C notation. In particular, the constructs `case_mode of`, `case_tag(T) of`, `if_var(T)`, `if_sva(T)` denote dispatching on mode or tag, testing whether T is instantiated to a variable or a stack variable, respectively. * is the memory reference operator, ++ is the postincrement operator, – is the predecrement operator.

The linear functions I and J perform trivial mappings from operand fields to word offsets, and are included for clarity.

3.1 Control Instructions

allocate(k)
This appears at the beginning of a clause which contains a procedure call followed by other instructions. Must be matched by a `deallocate` instruction. Space for a new environment is allocated on the stack.

```
CL := L; CE := E; E := A; A := A+I(k)
```

deallocate
This appears after computing the arguments of the last goal in a clause that has an `allocate(k)` instruction.

```
if A'≤E then A := E;
L := CL; E := CE
```

init(Xn)
This initializes Xn to a heap variable.

```
Xn := *(H++) := $var(H)
```

init(Yn)
This initializes Yn to a stack variable. The compiler emits these to guarantee that the current environent is fully initialized before the first `call` instruction of a clause. This measure is for protecting the garbage collector from dangling pointers.

```
Y := $sva(E+I(n)))
```

call(Label,k)
This corresponds to a procedure call that does not terminate a clause. The argument k is the number of variables in the environment at this point, and will be accessed by `cute` and `cutne` instructions where it is denoted by `env_size(L)`.

```
if A'≤E then A := E+I(k)
L := P; P := Label
```

execute(Label)
This corresponds to a procedure call that terminates a clause.

```
P := Label
```

proceed
This terminates a clause not terminated by a procedure call.

```
P := L
```

fail
This backtracks to the latest choicepoint.

```
P := $fail
```

3.2 Put Instructions

put_variable(Yn,Xi)
This represents a goal argument that is an unbound variable which occurs in the first general procedure call and later. After the first call instruction of a clause, the compiler emits `put_value Yn` instructions rather than `put_variable Yn`.

```
Xi := Yn := $sva(E+I(n)))
```

put_variable(Xn,Xi)
This represents a goal argument that is an unbound variable which does not occur after the next procedure call.

```
Xi := Xn := *(H++) := $var(H)
```

put_value(Vn,Xi)
This represents a goal argument that is a bound variable which cannot point (even before dereferencing) to a portion of the stack which is about to be deallocated.

```
Xi := Vn
```

put_unsafe_value(Yn,Xi)
This represents a goal argument that is a bound variable which might point to a portion of the stack which is about to be deallocated.

```
t1 := sderef(Yn);
if_sva(t1) then
  then if t1 >= E
       then (if A'>E then *(TR++) := t1);
            *(t1) := *(H++) := $var(H);
Xi := t1;
```

put_constant(C,Xi)
This represents a goal argument that is a constant.

```
Xi := C
```

put_nil(Xi)
This represents a goal argument that is nil.

```
Xi := $nil(0)
```

put_structure(F,Xi)
This represents a goal argument that is a structure.

```
Xi := $str(H); *(H++) := F; Mode := W
```

put_list(Xi)
This represents a goal argument that is a list.

```
Xi := $lst(H); Mode := W
```

3.3 Get Instructions

get_variable(Xn,Xi)
This represents a head argument that is an unbound variable.

```
Xn := Xi
```

get_variable(Yn,Xi)
This represents a head argument that is an unbound variable. The trail measure is necessary to prevent dangling pointer problems in garbage collection.

```
if A'>E then *(TR++) := Yn;
Yn := Xi
```

get_value(Vn,Xi)
This represents a head argument that is a bound variable.

```
unify(Vn,Xi);
```

get_constant(C,Xi)
This represents a head argument that is a constant.

```
unify(C,Xi)
```

get_nil(Xi)
This represents a head argument that is `nil`.

```
unify($nil(0),Xi)
```

get_structure(F,Xi)
This represents a head argument that is a structure.

```
t1 := deref(Xi);
case_tag(t1) of
  $var,$sva,$clo: unify(t1,$str(H));
                  *(H++) := F;
                  Mode := W
  $str: if *(t1++)=F
        then S := t1++1; Mode := R
        else fail
  otherwise: fail
```

get_list(Xi)
This represents a head argument that is a list.

```
t1 := deref(Xi);
case_tag(t1) of
  $var,$sva,$clo: unify(t1,$lst(H));
                  Mode := W
  $lst: S := t1;
        Mode := R
  otherwise: fail
```

3.4 Unify Instructions

unify_void(k)
This represents a sequence of k structure arguments that are singleton variables.

```
while k>0 do
  k := k-1;
  case_mode of
            R: t1 := *(S++);
            W: *(H++) := $var(H);
```

unify_variable(Xn)
This represents a structure argument that is an unbound variable.

```
case_mode of
  R: Xn := *(S++);
  W: Xn := *(H++) := $hva(H);
```

unify_variable(Yn)
This represents a structure argument that is an unbound variable. The trail measure is necessary to prevent dangling pointer problems in garbage collection.

```
if A'>E then *(TR) := Yn;
case_mode of
  R: Yn := *(S++);
  W: Yn := *(H++) := $hva(H);
```

unify_value(Vn)
This represents a structure argument that is a bound variable which cannot (even before dereferencing) be pointing to the stack.

```
case_mode of
  R: unify(Vn,*(S++));
  W: *(H++) := Vn
```

unify_local_value(Xn)
This represents a structure argument that is a bound variable which could be pointing to the stack.

```
case_mode of
  R: t1 := Xn; Xn := *(S++); unify(t1,Xn);
  W: t1 := sderef(Xn);
     if_sva(t1)
        then (if A'>E then *(TR++) := t1);
             Xn := $var(H++);
        else Xn := (H++) := t1;
```

The final dereferencing step is necessary since the compiler trusts the instruction to "globalize" its argument. Note that this cannot generally be done for permanent variables, because of backtracking. This instruction therefore has a slightly different semantics for permanent variables.

unify_local_value(Yn)
This represents a structure argument that is a bound variable which could be pointing to the stack.

```
case_mode of
  R: unify(Yn,*(S++));
  W: t1 := sderef(Yn);
     if_sva(t1)
       then (if A'>E then *(TR++) := t1);
            *(t1) := $var(H++);
       else *(H++) := t1
```

unify_constant(C)
This represents a structure argument that is a constant.

```
case_mode of
  R: unify(C,*(S++));
  W: *(H++) := C
```

unify_nil
This represents a structure argument that is `nil`.

```
case_mode of
  R: unify($nil(0),*(S++));
  W: *(H++) := $nil(0)
```

unify_structure(F)
This represents a last structure argument that is a structure.

```
case_mode of
  R: get_structure(F,*(S++));
  W: put_structure(F,*(H++))
```

unify_list
This represents a last structure argument that is a list.

```
case_mode of
  R: get_list(*(S++));
  W: put_list(*(H++))
```

3.5 Indexing Instructions

try(Proc,k)
This represents a first clause of arity *k* whose code starts at label *Proc*. The arity field will be accessed at backtracking, where it is denoted `arity_of(P')`.

```
Create a new choicepoint; P := Proc.
```

retry(Proc,k)
This represents a subsequent clause of arity *k* whose code starts at label *Proc*.

```
P' := P; P := Proc.
```

trust(Proc,k)
This represents a last clause of arity *k* whose code starts at label *Proc*.

```
B := B-J(k); P := Proc.
```

labels(Vl,Ll)
This represents a point in the code where one can branch to. At *Vl*, it is asserted that X0 is dereferenced to a variable. At *Ll*, it is asserted that X0 is dereferenced to the correct type matching the following get instruction.

switch_on_type(Va,Sy,St,Ni,Ls,In,Ch,Sg,Ot)
X0 is dereferenced, then a 9-branch switch is performed, depending on whether its tag is Variable, Symbol, Structure, Nil, List, Integer, Character, String, or Other.

switch_on_constant(Table,Default)
The *Table* is an alist of constants and labels. This instruction indexes on X0 to the appropriate label. If no value matches, branches to *Default*.

switch_on_structure(Table,Default)
The *Table* is an alist of functors and labels. This instruction indexes on X0 to the appropriate label. If no value matches, branches to *Default*.

3.6 The Cut Operator

The cut operator is a notorious problem for implementors. Its operational semantics is yet to be laid down firmly [Mos85].

We adopt the semantics of compiled DEC-10 Prolog, as formulated in [Per79]: "The cut operation commits the system to all choices made since the parent goal was invoked, and causes other alternatives to be discarded". By "the parent goal" we interpret the predicate *in which the cut textually appears*. Uses of cuts inside of meta-calls etc. are left undefined here, although it seems sensible to let such a cut be local to the meta-call.

Other variants of cut semantics include cuts that are local to disjunctions etc. We do not treat these, as they seem not strictly necessary, and less powerful than the DEC-10 version.

Achieving the desired semantics of cut is straightforward. The choicepoint register is saved away at procedure entry, and is restored by the cut operator.

Maintaining stack space economy is not as straightforward, however. The cut operator renders parts of the stack containing backtracking information inaccessible. These parts should preferably be reclaimed before the next environment or choicepoint is allocated. There could be dangling references into these parts from the trail, such references must be deleted too.

The problem with tidying the stack (in Warren's design) is that the garbage area is not always *visible*, i.e. a contiguous area at the stack top. It is not visible for cases when a predicate first allocates a choicepoint, then allocates an environment, then executes cut before deallocating. Now, the choicepoint is garbage but it is *buried* underneath an environment. The situation is even more complicated for cuts inside of disjunctions, where several environments can be active.

A partial answer to the deallocation problem is to have two separate stacks, one for environments and one for choicepoints. The answer is only partial, since garbage environments may still be buried after a cut. However, the split stack architecture causes other problems, and the tradeoffs are difficult to evaluate. For a more comprehensive discussion of the issues involved, see [Bar86c] or [Car86a].

choice(Vn)
This saves away the current choicepoint.

```
Vn := $sin(B)
```

cute(Vn)
This restores the current choicepoint, and tidies the trail.

```
B := $sva(Vn)

i := j := TR'
while i < TR do
```

```
    t1 := *(i++);
    case_tag(t1) of
      $sva: if t1<A' then *(j++) := t1;
      $var: if t1<H' then *(j++) := t1;
      $clo: if t1<H' then *(j++) := t1
            else if t1=*t1 then *(j++) := t1;
      otherwise: *(j++) := t1;
  TR := j
```

cutne(Vn)
This variant is used when no environment is active. It does all that `cute(Vn)` does, and adjusts the stack top:

```
cute(Vn);
A := max(A',E+env_size(L))
```

Note: `cute` and `cutne` first test whether the wake-flag has been raised, indicating that a delayed goal is pending, in which case the cut itself is delayed since the delayed goal could fail or introduce backtracking choices.

3.7 Utility Instructions

mode Functor(Modes
) The compiler simply passes mode declarations on to the code generator. A mode can be one of

'+': the corresponding argument is instantiated

'-': the corresponding argument is uninstantiated

'!': if the corresponding argument is uninstantiated, then delay on it

'?': (default), no information is given

order(Op,V1,V2)
V1 and *V2*, which are `temp(i)` or `perm(i)`, are compared acc. to op which is ==, \==, @<, @=<, @>, or @>=. The instruction causes failure if the test doesn't succeed. No argument registers are altered.

compute_fix_value(Vn,Expr,*live-X-vars***)**

compute_fix_variable(Vn,Expr,*live-X-vars***)**

compute_fix_constant(C,Expr,*live-X-vars***)**
Vn, which is `temp(i)` or `perm(i)`, or *C* is set to or unified with the value of expression *Expr*, which is built up from `perm(_)`, `temp(_)`, `constant(C)`, `structure(Op,Arglist)`. The live argument registers, listed in the third argument, are not altered.

compare_fix(Op,Args,*live-X-vars***)**
Op is a binary arithmetic operator. *Args* are Expressions as in `compute`. The instruction

causes failure if the test doesn't succeed. The live argument registers, listed in the third argument, are not altered.
This last instruction is not strictly necessary since it can be expressed in terms of `compute_fix_variable` and `order`.

3.8 Basic Operations

procedure call
At each procedure call, the F register is tested, and if nonzero, different actions are taken depending on its bits. Currently, the following actions are defined:

- waking delayed goals
- garbage collection when the heap is full
- tracing each predicate call
- counting each predicate call

deref(u)
Follow variable pointers from u until a fixpoint or a non-reference pointer is reached.

sderef(u)
Follow stack variable pointers from u until a fixpoint or a non-stack reference pointer is reached.

fail
Unwind trail as far as TR'. Reset wake-bit of F. Restore H, E, L, A, and X0..Xm, where `m=arity_of(P')`. Branch to P' which is asserted to be a `retry` or a `trust` instruction.

```
while TR>TR' do
  t1 := *(--TR);
  if_var(t1)
    then *(t1) := t1
    otherwise: "handle exception"

F<wake> := 0; H := H'; E := E'; L := L'; A := A';
X0..Xm := X0'..Xm'; P := P'
```

unify(x,y)
Dereference x and y and unify them. Variable-variable instantiations obey the convention to always instantiate the variable with the higher address.
If a `$clo` variable is instantiated, the following action is taken:

```
Let G be the suspended goal.
if F<wake>
   then W := (W,G)
   else W := G;
        F<wake> := true
```

4. Kernel Predicates

4.1 The Interpreter

Although the implementation is based on compilation, we have included an interpreter to support program debugging. The interpreter consists of a meta-interpreter in Prolog and certain built-in primitives. The meta-interpreter exists in a basic version, for normal interpretation, and in an extended version, for debugging.

Every functor points to executable code: `$fail` (`$error`, eventually) for undefined predicates, compiled code for compiled predicates, and an interface to the interpreter for interpreted predicates. This interface packs the functor and arguments into a term and executes the toplevel predicate of the meta-interpreter, whose pre-defined name is `$interpret_goal/1`.

The predicate `call/1` spreads the argument and just executes the code pointed to by the functor. For interpreted predicates, it just executes `$interpret_goal/1`.

Every interpreted predicate points to a list of *clause objects*. A clause object is conceptually a byte code representation of abstract machine code for a unit clause of the predicate `clause/2`.

The interpreter consists of a meta-interpreter which is supported by built-in primitives which insert or delete clause objects under functors, retrieve the list of clauses for the proper functor, and emulate coded clause objects.

4.2 Delay Primitives

The machinery for delaying goals is largely implemented by special code emitted by mode declarations and by unification. Suspension objects are created by the built-in primitive `$geler`. They are trailed at creation time, so that all suspensions can be found by traversing the trail.

$geler
(X0: variable, X1: goal).

```
case_tag(X0) of:
     $var:   *(H++) := t1 := $clo(H);
             *(H++) := X1;
             *(TR++) := t1;
             unify(X0,t1);

     $clo:   *(H++) := t1 := $clo(H);
             *(H++) := $str(H++1);
             *(H++) := $fnt(',',2);
             *(H++) := *(X0++1);
             *(H++) := X1;
             *(TR++) := t1;
```

```
            unify(X0,t1);

      oterwise: fail.
```

The toplevel checks the list of suspensions to see whether any suspensions still have not been run. This check should be done in other situations as well, in particular by negation and all solution predicates.

The $geler mechanism delays a goal until a variable binding occurs, but a more useful behavior is to delay until the principle functor of the variable is known. This is easily built upon $geler as follows:

```
freeze(Var,Goal) :-
   var(Var), syscall('$geler',[Var,freeze(Var,Goal)]);
   nonvar(Var), Goal.
```

or, equivalently:

```
:- mode freeze(!,?).
freeze(_,G) :- G.
```

The choice of $geler rather than freeze/2 as the underlying primitive is motivated by the implementation of sound inequality as

```
X≠Y :-
   syscall('$dif',[X,Y,Z1,_]),
   (var(Z1), !, syscall('$geler',[Z1,X≠Y]);
    true).
```

where $dif succeeds unless X0 and X1 are identical terms, returning in X2 and X3 the first subterms of X0 and X1 where a difference is established, and X2@<X3.

Delaying on more than one variable is achieved as follows:

```
freeze_list(Vas,Goal) :-
   freeze_list_1(Vas,freeze_list_2(_,Goal)), !;
   Goal.

freeze_list_1([],_).
freeze_list_1([Va|Vas],Goal) :-
   var(Va), freeze(Va,Goal), freeze_list_1(Vas,Goal).

freeze_list_2(M,Goal) :-
   nonvar(M);
   var(M), M=1, Goal.
```

Delaying until a term is ground (upon which sound negation is built) is implemented as follows:

```
freeze(Goal) :-
```

```
    contains_variable(Goal,Var), !, syscall('$geler',[Var,freeze(Goal)]);
    Goal.

contains_variable(X,X) :- var(X).
contains_variable(Struct,X) :-
    arg(1,Struct,Arg),
    arg_contains_variable(Struct,X,1,Arg).

arg_contains_variable(Struct,X,I,Arg) :-
    contains_variable(Arg,X);
    I1 is I+1,
    arg(I1,Struct,Arg1),
    arg_contains_variable(Struct,X,I1,Arg1).
```

5. Overview of Compilation

5.1 Introduction

The compilation proceeds in two file-to-file steps:

PLWAM
The Prolog source code is compiled to machine-independent WAM (Warren Abstract Machine) code.

WAMTQL
The WAM code is compiled to native machine code, stored in TQL (Tricia Quick Load) format.

The PLWAM step has the following phases (roughly), repeated for each predicate:

1. Collect mode declaration and clauses
2. Convert source code to record structures
3. Spawn internal predicates
4. Allocate permanent variables (per clause)
5. Process mode declarations and cut (per predicate)
6. Emit instructions (per clause)
7. Analyze lifetimes of temporary variables (per clause)
8. Allocate temporary variables (per clause)
9. Emit linkage instructions (per predicate)
10. Do peep hole optimization and final editing

Each phase is discussed in more detail below.

5.2 Collect mode declaration and clauses

The clauses that comprise a predicate, optionally preceded by a mode declaration, are compiled together. This causes a space problem for large predicates, which could be solved by having a separate clause compiler.

The mode declaration affects the semantics as follows:

With a '+' declaration on the first argument, the predicate will fail if called with the argument uninstantiated.

With a '-' declaration on the first argument, the predicate will fail if called with the argument instantiated.

With a '!' declaration on any argument, the predicate will suspend if called with the corresponding argument uninstantiated.

5.3 Convert source code to record structures

In order to minimize the use in the compiler of meta-logical predicates, the source code is converted to more manageable records.

var(X)
This record type denotes a source variable.

constant(C)
This record type denotes a ground source term.

nil
This record type denotes the source term `[]`.

list(X,Y)
This record type denotes the source list `[x|y]`. X and Y are records denoting the arguments x and y.

structure(S,L).
This record type denotes a source structure. S is the functor and L is a list of records denoting the arguments of the structure.

This phase specially recognizes cuts, and transforms them to calls `$cut(B)`, where it is arranged so that B is the current choicepoint at procedure entry.

5.4 Spawn internal predicates

This phase recognizes certain situations where pieces of code are broken off as internal predicates and are replaced by calls to these. The situations are:

sub-argument indexing
Groups of clauses where the first argument have the same principal functor are broken off as a predicate indexed on the first sub-argument. The process is repeated recursively.
Example. The predicate:

```
p(f(X)) :- f1(X).
p([1|X],X).
p([2|X],Y) :- q(X,Y).
p(f(a,X,Y),X,Y).
p(f(b,X,Y),Y,X).
p(g(X)) :- g1(X).
```

This is transformed to:

```
p(f(X)) :- f1(X).
p([X|Xs],Y) :- p'(X,Y,Xs).
p(f(A,B,C),D,E) :- p"(A,D,E,B,C).
p(g(X)) :- g1(X).

p'(1,X,X).
```

```
p'(2,Y,X) :- q(X,Y).

p"(a,X,Y,X,Y).
p"(b,X,Y,Y,X).
```

disjunction

A disjunction of N disjuncts is broken off as a predicate with N clauses. However, a clause which contains materially nothing but a disjunction is replaced by N clauses.

Example. The predicate:

```
p([A|B],C) :- f(C,D), (g1(A,D); g2(B,D)).
p(A,B) :- h1(A); h2(B).
```

This is transformed to:

```
p([A|B],C) :- f(C,D), p'(A,B,D).
p(A,_) :- h1(A).
p(_,B) :- h2(B).

p'(A,_,D) :- g1(A,D).
p'(_,B,D) :- g2(B,D).
```

negation

Negation as failure is always broken off as an internal predicate.

Example. The predicate:

```
p([A|B],C) :- f(C,D), \+ g(A,[B|D]).
```

This is transformed to:

```
p([A|B],C) :- f(C,D), p'(A,B,D).

p'(A,B,C) :- g(A,[B|C]), !, fail.
p'(_,_,_).
```

if_then_else

If_then_else by test and commitment is always broken off as an internal predicate.

Example. The predicate:

```
p([A|B],C) :- f(C,D), (t(A) -> q(B); q(D)).
```

This is transformed to:

```
p([A|B],C) :- f(C,D), p'(A,B,D).

p'(A,B,_) :- t(A), !, q(B).
p'(_,_,C) :- q(C).
```

Discussion

Each spawned internal predicate is treated by the compiler just as a user predicate. This strategy of compiling disjunctions has the advantage that there are no branches in the code for a clause, which greatly simplifies lifetime analysis. Updating variable descriptors becomes straightforward also.

This is in contrast to the approach taken e.g. in Van Roy's compiler [Roy84], where disjunctions are coded in-line. Van Roy avoids sometimes having to allocate extra environments, but the compile time analyses become complicated by the code being non-linear, and the implementation of cut runs into some extra trouble.

In general, transformations such as the above are perhaps better handled by a precompiler. A precompiler could do more extensive transformations such as partial evaluation in a more comprehensive manner.

Of course, the above variants of negation and if_then_else are not sound. Sound versions using delay primitives should be provided as well.

5.5 Allocate permanent variables

A variable is *permanent* if it has next use after a procedure call, otherwise it is *temporary*.

Warren additionally classifies a variable which occurs first as an argument in a procedure call and occurs last in the same call as permanent.

Permanent variables are arranged such that they can be discarded as soon as possible. This is done by walking each clause from right to left but each goal from left to right, and assigning each new permanent variables numbers in increasing order.

Permanent variables that have 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 extra optimization has not yet been added.

5.6 Process mode declarations and cut

The code for each internal predicate starts with a prelude of up to 2 instructions:

mode Modes

if there is an explicit mode declaration,

choice(*Xn*)

if the predicate lexically contains a cut. Xn is the first free argument register. The arity of the predicate is effectively increased by one.

Example: The predicate

```
p(X,Y) :- s(X), !, t(Y).
```

is effectively treated as:

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

```
p"(X,Y,Z) :- s(X), $cut(Z), t(Y).
```

although here p" is not broken off as a separate internal predicate.

5.7 Emit instructions

Each clause is translated to a sequence of instructions in a fairly naive way. No `allocate` or `deallocate` instructions are emitted here, for simplicity. Temporary variables are left unallocated.

The head arguments are not compiled in textual order. Instead, temporary variables are compiled first. Voids are ignored altogether. The motivation behind this heuristic is twofold. First, try to free machine registers for use as temporaries which the code for complex arguments might need. Second, try to condense sequences like

```
unify_variable(Xm)
...
get_value(Xm,An)
```

into

```
get_variable(Xm,An)
...
unify_local_value(Xm)
```

In the latter sequence, `Xm` and `An` can always be assigned to the same temporary register, rendering the `get_variable` instruction superfluous.

Conversely, goal arguments are not compiled in textual order. Complex arguments are compiled first. The rationale is that complex arguments may need temporary registers, therefore these are kept free as long as possible.

Descriptors are maintained for X register contents. They are cleared by call instructions, and updated by all other instructions. References to constants and to temporary variables are replaced by references to X registers, when the X register is known to contain the right value. X register assignments are omitted when the X register is known to already contain the correct value. This is not always optimal, since not all temporary variables reside in machine registers. Replacing references to constants or permanent variables by references to temporary variables may also extend the lifespans of temporary variables.

Descriptors are maintained for all variables. A variable can be in one of the following states:

uninitialized
The variable does not have a value yet.

local
The variable could point to the current environment.

remote
The variable could point to the stack, but only below the current environment.

global
The variable cannot point to the stack.

These descriptors are consulted when deciding whether `unify_local_value` is required or `unify_value` (which is cheaper) suffices, and whether `put_unsafe_value` is required or `put_value` (which is cheaper) suffices, and whether `*_variable` or `*_value` instructions should be emitted.

The descriptors are updated by certain instructions. The updating scheme must be very carefully tuned to the exact semantics of `unify_local_value`, `unify_value`, `put_unsafe_value`, and `put_value` instructions for correctness to be preserved.

5.8 Analyze lifetimes of temporary variables

For each instruction in a translated clause, a list of the temporary variables with next use after the instruction is computed. This is done by the standard technique of walking the code in reverse order, adding or deleting such temporaries from the current list that are used or defined, respectively. Calls are to considered to 'use' argument registers up to the arity of the called predicate. Calls are considered to destroy i.e. 'define' all temporaries. This is certainly overkill for calls to built-in predicates. For simplicity and modularity, such calls are not presently treated specially.

5.9 Allocate temporary variables

Temporary variables must be allocated such that all next_use lists are free from duplicates. A duplicate would correspond to an overloaded temporary variable. This forms a simple correctness criterion on temporary variable allocation.

It is desirable to identify null operations such as

```
get_variable Xi,Ai
put_value Xi,Ai
```

since such vacuous instructions can be deleted. This forms a quality measure on variable allocation.

It is desirable to maximize the use of such X registers that reside in machine registers. This forms another quality measure on variable allocation.

The compiler does not attempt to achieve an optimal allocation. The following algorithm is used:

1. For each `get_variable` or `put_value` instruction, equate its arguments unless this creates a duplicate in some next_use list.
2. For each yet unallocated temporary variable, assign it to the lowest possible X register number which does not create a duplicate in any next_use list.

This abstract algorithm actually proceeds in smaller steps to avoid as much search of next_use lists as possible. Specializations of the algorithm are discussed in [Car86b].

Similar algorithms have been developed by Van Roy [Roy84] and by Debray [Deb86], although in Debray's compiler, code generation and register allocation are intertwined.

5.10 Emit linkage instructions

Linkage instructions provide pathways from a predicate's entrypoint to individual clauses using indexing on the first argument where possible.

In Warren's original design, clauses are arranged in groups, where a group is either a collection of clauses with a non-variable first argument, or a single clause with a variable first argument. The indexing instructions try each group in turn. For non-variable groups, indexing is performed on the principal functor of the first argument. Thus the method may create two choicepoints before entering a clause.

In our method, all clauses are treated as a single group, and indexing is done at once on the principal functor of the first argument yielding a set of potentially matching clauses whose first argument is either the same principal functor or a variable. A fuller description of the method and a comparison with other methods can be found in [Car86a].

5.11 Peep hole optimization and final editing

This phase traverses the entire code for all internal predicates and does necessary final editing as follows:

1. `allocate` and `deallocate` instructions are inserted where appropriate. `allocate` instructions are inserted as late as possible, i.e. just before the first call or permanent variable reference in a clause in order to possibly fail before allocating.
2. `init Yn` instructions are emitted before the first `call`, to ensure that the environment is fully initialized. This measure is necessary to prevent the possibility of dangling pointers in the garbage collector, since it will traverse all environments in its mark phase. The measure also enables the strength reduction of `put_variable Yn` instructions after the first `call` to `put_value Yn` instructions.
3. Vacuous `get_variable U,U` and `put_value U,U` instructions are deleted.
4. Next_use information is filled in for instructions which require it, for the sake of generating machine code for arithmetic expressions.

Some other optimizations could be performed here as well, like condensing sequences of `unify_void` instructions. This is presently done already while emitting instructions for unification.

6. Conclusions

We have described an elaboration of the Warren Abstract Machine for Prolog along with its compiler. The elaborations include arithmetic, cut and delay operators, and constraints imposed by a garbage collector. Various compiler optimizations were introduced and discussed.

We conclude that it is hard to generate optimal sequences of abstract machine code, but that a few heuristics yield very reasonable code.

The Warren abstract machine is an extremely good target language for a Prolog compiler. Operationally, however, the various instructions differ wildly in complexity: some are mere transfers of values, others are unbounded in time. An interesting issue for future research is to try to find an instruction set which bridges over the semantic gap betweeen Warren's abstract machine code, and the native machine code of typical processors.

7. Acknowledgments

The work reported herein was carried out in part at UPMAIL, in part during a visit at the Weizmann Institute of Science, Israel, and in part at the Swedish Institute of Computer Science (SICS). The Computing Science department of Uppsala University supplied the computer resources. The work was supported in part by the National Swedish Board for Technical Development (STU).

The results reported herein would not have been possible without the unique research environment at UPMAIL. We stress that the final design of the instruction set is the result of a team work and several iterations involving J. Barklund's code generator [Bar86a] and H. Millroth's garbage collector [Bar86b,Bar86cc]. Many aspects of the instruction set, especially those involving arithmetic and clause indexing, were worked out in (sometimes heated) discussions with J. Barklund.

Intellectually, we owe a lot to David H. D. Warren who invented the original "New Engine" concept, upon which this work builds.

It is difficult to give a precise account of each person's contribution to the implementation, but the following is a close approximation:

J. Barklund implemented the PDP-10 code generator, the interpreter, related parts of the runtime system, the system kernel and wrote the garbage collection algorithm.

M. Carlsson wrote the compiler and adapted part of the Prolog runtime system, obtained from public domain sources, to Tricia.

H. Millroth implemented the garbage collector and substantial parts of the runtime system.

Lennart Gidlund, Bo Arvidson, Torbjörn Åhs, Åsa Hugosson, Mats Nylén and Lars Oestreicher contributed substantial parts of the implementation and documentation.

Thanks also to Sverker Janson, Thomas Sjöland, Seif Haridi, Jaakov Levy, Lee Naish, and Ross Overbeek for useful criticism, numerous discussions, and sharing important insights.

Appendix A. References

[Bar86a] Barklund J., Millroth H., "Code Generation and Runtime System for Tricia", UPMAIL Technical Report no. 36, 1986.

[Bar86b] Barklund J., Millroth H., "A Garbage Collection Algorithm for Tricia" UPMAIL Technical Report no. 37, 1986.

[Bar86c] Barklund J., Millroth M., "Garbage Cut for Garbage Collection of Iterative Prolog Programs", IEEE Symposium for Logic Programming, Salt Lake City, Utah, 1986.

[Car86a] Carlsson M., "On Indexing and Cut in the WAM", SICS Research Report, Swedish Institute of Computer Science, 1986 (in preparation).

[Car86b] Carlsson M., "Temporary Variable Allocation for the WAM", SICS Research Report, Swedish Institute of Computer Science, 1986 (in preparation).

[Deb86] Debray S., "Register Allocation for a Prolog Machine", IEEE Symposium for Logic Programming, Salt Lake City, Utah, 1986.

[Mos85] Moss C., "Investigating implementations of cut", *Prolog Digest*, Vol. 3 #30, Usenet, 1985.

[Nai85] Naish L., "Negation and Control in PROLOG", Ph.D. Thesis, Department of Computer Science, University of Melbourne, 1985.

[Per79] Pereira L., Byrd L., Pereira F., Warren D.H.D., "User's Guide to DECsystem-10 Prolog", DAI Occasional Paper 15, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, 1979.

[Roy84] Van Roy, "A Prolog Compiler for the PLM", M.Sc. Thesis, Department of Computer Science, University of California at Berkeley, 1984.

[War83] Warren D.H.D., "An Abstract Prolog Instruction Set", SRI International #309, 1983.