Logic for Systems Note 1: Extending the WAM with Procedure Box Debugging

Lindsey Spratt
March, 2019.

Abstract

A trace and interactive debug facility organized around the procedure box control flow model is a powerful addition to a Prolog implementation. Implementing this facility is complicated when the Prolog implementation is built around the Warren Abstract Machine (WAM). This note describes in detail an approach to extending a standard WAM implementation to support a trace and interactive debug facility.

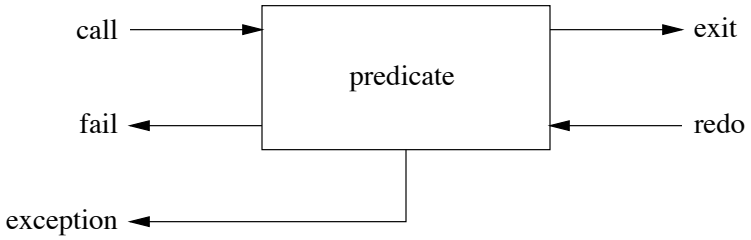
Tools for debugging Prolog are essential for a useable implementation. The standard approach is powerful bug complicated to implement in an integrated way with the Warren Abstract Machine (WAM). This note provides a detailed tutorial approach to such an implementation extending the final Prolog WAM model in [Aït-Kaci, 1999]. In this note we present a simple trace and debugging implementation as a series of simplified Prolog meta-interpreters then follow that with the WAM-based implementations. First we present the standard debugging model.

The Procedure Box Control Flow Model

The standard model for tracing and debugging Prolog is the procedure box control flow model introduced by Lawrence Byrd [Byrd 1980]. Some version of this model is available in

all mature Prolog implementations. The GNU Prolog manual describes this model as follows:

> The debugger executes a program step by step tracing an invocation to a predicate (call) and the return from this predicate due to either a success (exit) or a failure (fail). When a failure occurs the execution backtracks to the last predicate with an alternative clause. The predicate is then re- invoked (redo). Another source of change of the control flow is due to exceptions. When an exception is raised from a predicate (exception) by throw/1 [...] the control is given back to the most recent predicate that has defined a handler to recover this exception using catch/3 [...]. The procedure box model shows these different changes in the control flow, as illustrated here:



> Each arrow corresponds to a port. An arrow to the box indicates that the control is given to this predicate while an arrow from the box indicates that the control is given back from the procedure. This model visualizes the control flow through these five ports and the connections between the boxes associated with subgoals. Finally, it should be clear that a box is associated with one invocation of a given predicate. In particular, a recursive predicate will give raise to a box for each invocation of the predicate with different entries/exits in the control flow. Since this might get confusing for the user, the debugger associates with each box a unique identifier (i.e. the invocation number). (p. 31, [Diaz, 2013])

SICStus Prolog also uses this five-port execution model. SWI-Prolog implements a six-port version that adds a unify port: “The additional unify port allows the user to inspect the

result after unification of the head" (p. 29, [Wielemaker 2019]). XSB Prolog implements the basic four-port debugger (p. 315, [Swift et al, 2013]). Ciao Prolog also implements the basic four-port debugger ([Cabeza et al, 2018]).

Some of these Prolog implementations have much more sophisticated debugging environments than the command-line procedure box control flow approach. However this is in addition to a command line interaction for the procedure box as a basic debugging tool.

This document focuses on how to support command-line interactive control of the four-port version with the basic set of call, exit, fail, and redo. The interaction occurs as the Prolog evaluation arrives at a port of the procedure box where supported commands include: "c" (creep), "s" (skip), "l" (leap), "+" (spy this), "-" (nospy this), "f" (fail), "r" (retry), "g" (ancestors), "a" (abort), and "n" (nodebug).

This model can be implemented by creating an interpreter in Prolog for Prolog programs and instrumenting this interpreter appropriately. Such an interpreter relies on the clause/2 predicate to discover the clauses that define a predicate. This approach does not work with Prolog programs that have been compiled to use the WAM since the clause/2 predicate does not have access to the clauses for the compiled predicates. This is because the compiled predicate's clauses are stored only as byte sequences specifying instructions that are to be evaluated by the WAM: the logical representations of the clauses are not retained.

Mature Prolog systems provide some version of tracing and debugging support that is integrated with their implementation of the WAM. There does not appear to be any generally available discussion of how to achieve this integration.

This paper describes how the four-port model for tracing can be supported directly in the WAM. Hassan Aït-Kaci's *Warren's Abstract Machine: A Tutorial Reconstruction* [Aït-Kaci, 1991] provides an elegant explanation of the

implementation of the WAM.[1]The final model of this book is extended in this note with elements needed to integrate tracing with interactive controls. The approach to tracing presented here was inspired by the implementation of GNU Prolog [Diaz, 2013]. The approach was tested in an implementation extending Matt Lilley's Proscript project https://github.com/thetrime/proscript in a fork by the author at https://github.com/lindseyspratt/proscript .

The $trace predicate

The basic implementation scheme is to conditionally divert the CALL and EXECUTE instructions for a predicate-to-be-traced to invoke a '$trace' predicate with the to-be-traced predicate and its arguments as Prolog structure term in an argument to '$trace'. The '$trace' predicate is written in Prolog and compiled. It manages the user interaction: displays the trace information, gets the user's command, continues the WAM execution of the subject predicate call using the Prolog call/1 builtin (after setting a flag to avoid immediately re-invoking the '$trace' predicate on this same subject predicate call), and following that call with more user interaction management.

A general version of this predicate is '$trace'/1.

```
'$trace'(Goal) :-
  '$trace_interact'('Call',
    'Fail', Goal),
  call(Goal),
  '$trace_set'(no_trace),
  '$trace_interact'('Exit',
    'Redo', Goal).
```

The '$trace_set' builtin sets a flag that the WAM inspects to determine when to invoke the '$trace' predicate. The '$trace_interact' predicate when 'creeping' on a 'call' port sets the trace mode to trace_next_jmp: this tells call/1 and the WAM to active tracing for the *next* call or execute instruction. When '$trace_interact' is 'creeping' on an 'exit' port it sets the trace mode to 'trace' as its final predicate call. Setting the mode to no_trace after 'call(Goal)' tells the WAM to not invoke '$trace' so that the call of '$trace_interact'/3 and any predicates it may call are not (recursively) traced. The call of '$trace_set' is itself a risk of recursively invoking the '$trace' predicate: this is avoided by an explicit prohibition in the WAM call and execute instructions. They never invoke '$trace' on '$trace_set' regardless of the current

[1] David S. Warren's "WAM for Everyone" [Warren, 2018] is a more modern and succinct presentation of the WAM. However it is not used as a reference point in this document because it does not provide the procedural detail that [Aït-Kaci, 1999] presentation does.

trace mode. (This prohibition may be extended to not invoking '$trace' on any predicate with a functor starting with '$trace'.)

Using the '$trace' predicate approach allows us to express most of the trace and debug logic in Prolog with only small modifications to the WAM implementation.

The following material presents the tracing features in a simple Prolog interpreter written in Prolog then presents the implementation of these features in modifications to the final Prolog-like WAM instructions in [Aït-Kaci, 1999] and an associated '$trace' predicate. There are several versions of the trace/debug implementation, each one adding more features.

A Tracing Prolog Interpreter

The basic 4-port procedure box control flow model can be demonstrated using a Prolog interpreter written in Prolog. The following discussion develops a series of interpreters, interpret1/1 through interpret4/1, with progressively more informative tracing. A fifth version, interpret5/1, is presented later that introduces user interactions.

interpret1/1: simple interpreter without tracing.

The most basic interpreter. interpret1/1:

This interpreter uses the cls/2 predicate to get program clauses.

```
interpret1(true) :- !.
interpret1((G1, G2)) :- !,
interpret1(G1), interpret1(G2).
interpret(Goal) :-
  cls(Goal, MoreGoals),
  interpret1(MoreGoals).
```

The simple program for predicates p/0, q/0, and r/0 is shown where p/0 is true if q/0 is true or if r/0 is true and each of q/0 and r/0 are true as simple facts.

```
p :- q.
p :- r.
q.
r.
```

This program is represented using the cls/2 predicate to make it evaluatable by interpret1/1.

```
cls(p, q).
cls(p, r).
cls(q, true).
cls(r, true).
```

Evaluating interpret1(p) succeeds twice (once for q and once for r).

interpret2/1: extending to basic four-port display

We can extend interpret1 to display the 4-port procedure box control flow model as interpret2/1.

```
interpret2(true) :- !.
interpret2((G1, G2)) :- !,
interpret2(G1), interpret2(G2).
interpret2(Goal) :-
  msg('Call', 'Fail', Goal),
  clause(Goal, MoreGoals),
  interpret2(MoreGoals),
  msg('Exit', 'Redo', Goal).
```

This program is very similar to interpret1 with the third clause (for interpret2(Goal)) extended using the new msg/3 predicate to display the trace information. Evaluating interpret1(p) and interpret2(p) in gprolog produces:

```
| ?- interpret1(p).

true ? ;

yes
| ?- interpret2(p).
```

```
msg(Success, _Failure, Goal) :-
  write(Success), write(' '),
  write(Goal), nl.

msg(_Success, Failure, Goal) :-
  write(Failure), write(' '),
  write(Goal), nl, !, fail.
```

```
Call p
Call q
Exit q
Exit p

true ? ;
Redo p
Redo q
Fail q
Call r
Exit r
Exit p

true ? ;
Redo p
Redo r
Fail r
Fail p

no
| ?-
```

In this console output we can see that interpret1(p) succeeds twice.
The interpret2(p) evaluation displays three results, the first succeeds using ‘q’, the second redoes p and q then succeeds using r, and the third result redoes p and r then fails p.

The trace facility in a prolog such as GNU Prolog or SWI-Prolog has a more informative display and provides interactive commands that allows the developer to explore the evaluation in detail. We can extend interpreter2 to provide a more informative display by showing the stack depth of a particular goal evaluation and a unique identifier of each evaluation.

interpret3/2: extending to display evaluation depth

Extending the interpreter to show the evaluation depth is shown in interpret3/2.

The second argument of interpret3/2 is a list of the goals that are ancestors of the evaluation of the current goal. msg3/3 is an extension of msg2/3 that displays the number ancestors for the current goal as the evaluation depth. A trace of interpret3(p, []) is:

```
| ?- interpret3(p, []).
0 Call p
1 Call q
1 Exit q
0 Exit p

true ? ;
0 Redo p
1 Redo q
1 Fail q
1 Call r
1 Exit r
0 Exit p

true ? ;
0 Redo p
1 Redo r
1 Fail r
0 Fail p

no
```

In this trace the ‘0 Call p’ shows the top level evaluation of p/0 which has no ancestors and so is at depth 0, followed by the evaluation of q/0 at depth1. The user requests another answer (‘;’) forcing the ‘Redo’ of p and q and the evaluation of r. The user again requests another answer which fails.

```
interpret3(true, _) :- !.

interpret3((G1, G2), Anc) :-
  !,
  interpret3(G1, Ancestors),
  interpret3(G2, Ancestors).

interpret3(Goal, Anc) :-
  NewAncestors = [Goal|Anc],
  msg3('Call', 'Fail', NewAnc),
  cls(Goal, MoreGoals),
  interpret3(MoreGoals, NewAnc),
  msg3('Exit', 'Redo', NewAnc).


msg3(Success, _, [Goal|Anc]) :-
  length(Anc, Depth),
  write(Depth), write(' '),
  write(Success), write(' '),
  write(Goal), nl.

msg3(_, Failure,[Goal|Anc]) :-
  length(Anc, Depth),
  write(Depth), write(' '),
  write(Failure), write(' '),
  write(Goal), nl, !, fail.
```

interpret4/1: extending to include invocation identifier

We add the invocation identifier and its display in interpreter4:

The invocation/1 predicate is used to store the current invocation number persistently across backtracking of failures by retracting the old fact and asserting a new fact with the updated invocation number. (The retract/1, retractall/1, and assertz/1 builtin predicates do not undo their clause base modifications on backtracking.)

Using interpret4(p) we get:

```
| ?- interpret4(p).
      1    1  Call: interpret4(p) ? l
1 0: Call p
2 1: Call q
2 1: Exit q
1 0: Exit p

true ? ;
1 0: Redo p
2 1: Redo q
2 1: Fail q
3 1: Call r
3 1: Exit r
1 0: Exit p

true ? ;
1 0: Redo p
3 1: Redo r
3 1: Fail r
1 0: Fail p

no
```

The first column is the invocation number and the second column is the depth indicator (as in interpret3/2).

```
interpret4(Goal) :-
  clear_invocation,
  interpret4(Goal, []).


interpret4(true, _) :- !.

interpret4((G1, G2), Anc) :-
  !,
  interpret4(G1, Anc),
  interpret4(G2, Anc).

interpret4(Goal, Anc) :-
  increment_invocation(K),
  NewAnc = [Goal|Anc],
  !,
  msg4('Call', 'Fail',
    NewAnc, K),
  cls(Goal, MoreGoals),
  interpret4(MoreGoals, NewAnc),
  msg4('Exit', 'Redo',
    NewAnc, K).


msg4(Success, _Failure,
    [Goal|Anc], Invocation) :-
  length(Anc, Depth),
  write(Invocation), write(' '),
  write(Depth), write(': '),
  write(Success), write(' '),
  write(Goal), nl.

msg4(_Success, Failure,
    [Goal|Anc], Invocation) :-
  length(Anc, Depth),
  write(Invocation), write(' '),
  write(Depth), write(': '),
  write(Failure), write(' '),
  write(Goal), nl, !, fail.


clear_invocation :-
  retractall(invocation(_)),
  assertz(invocation(0)).


increment_invocation(K) :-
  retract(invocation(J)),
  K is J + 1,
  assertz(invocation(K)).
```

Integrating Tracing with the WAM

This simple interpreter demonstrates the basic information that we would like to display and it provides a framework that can be extended in many ways to support interactions, ancestor display, spying on specific predicates (much like setting break points in other debugging models), and more. The complication is in integrating this with the WAM byte interpreter.

The specific problem solved here is how to extend the final WAM of [Aït-Kaci, 1999] to support the sort of trace output demonstrated above by the interpret4/1 predicate.

The design goal is to modify the WAM as little as possible and implement the bulk of the trace behavior in Prolog code that cooperates with these WAM modifications.

One major complication is that there is no version of the cls/2 predicate (e.g. clause/2) available to us that reveals the clauses implementing a predicate once that predicate has been compiled. The information about what are the clauses of a compiled program is encoded in WAM instructions (in GNU Prolog this encoding may be as machine instructions). We need to integrate the trace behavior with various of these WAM instructions: call, execute, try_me_else, retry_me_else, and trust_me.

The simplest trace mechanism

This first version of the trace mechanism corresponds to the interpret2/1 example above. The displayed trace information is only Call, Exit, Fail, and Redo for each goal.

The basic design of the trace mechanism is to have the **call** ***P*** and **execute** ***P*** instructions conditionally translate to **call** '$trace'/1 and **execute** '$trace'/1 instructions where the A_1 register is a structure of functor(***P***) with arguments from the original A_i.

There are two new registers in the WAM_t: TC, the trace call register, and TP, the trace predicate register. The TC register indicates what should be done regarding tracing of call and execute instructions: 0 is *no_trace*, 1 is *trace*, 2 is *trace_next*, and 3 is *trace_next_jmp*.

The TP register is the code address of the compiled code for the ‘$trace’/1 predicate. The TP register is set on initialization of the WAM and never changes. The ‘$trace’/1 predicate provides the actual tracing output for a call of a goal.

The choice-point frame is extended with one more slot for TC. This makes 8 fixed slots in the choice-point frame instead of 7.

The ‘$trace’/1 predicate.

The bulk of the trace logic is implemented in Prolog code, the ‘$trace’/1 predicate that produces the simplest tracing, analogous to the interpret2/1 predicate discussed above.

```
‘$trace’(notrace) :-
  !,
  ‘$trace_set’(no_trace).

'$trace'(Goal) :-
  '$trace_set'(no_trace),
  '$trace_msg'(‘Call’, ‘Fail’,
    Goal),
  '$trace_set'(trace_next_jmp),
  call(Goal),
  '$trace_set'(no_trace),
  '$trace_msg'(‘Exit’, ‘Redo’,
    Goal),
  '$trace_set'(trace).

'$trace_msg'(Success, _, Goal) :-
  ‘$trace_msg1’(Success, Goal).
'$trace_msg'(_, Failure, Goal) :-
  ‘$trace_msg1’(Failure, Goal),
  !,
  fail.

‘$trace_msg1’(Label, Goal) :-
  write(Label),
  write(''),
  writeln(Goal).
```

The ‘$trace_set’/1 builtin predicate sets the WAM TC register directly. It invokes the trace_set procedure.

The intent of the call/1 goal in ‘$trace’/1 is to evaluate the traced Goal without (recursively) invoking ‘$trace’/1 on it. The management of the TC register achieves this intent. This implementation allows for the call/1 predicate itself to be implemented in Prolog and to rely on a ‘$jmp’/1 builtin predicate to make the final preparations for redirecting the WAM to evaluate the instructions for Goal.

```
procedure trace_set(mode);
  begin
    if (mode = ‘no_trace’)
      then TC <- 0
    else if (mode = ‘trace’)
      then TC <- 1
    else if (mode = ‘trace_next’)
      then TC <- 2
    else if
        (mode = ‘trace_next_jmp’)
      then TC <- 3
    else ERROR;
  end;
```

Setting TC == *trace_next_jmp* makes the '$jmp'/0 predicate set TC == *trace_next*. The '$jmp'/0 predicate is used in the implementation of the call/1 predicate to prepare for the invocation of the WAM **call** instruction. The *trace_next* mode makes the WAM call instruction set TC == *trace* so that the *next* evaluation of the **call** instruction will invoke '$trace'/1.

```
trace :-
  '$trace_set'(trace).

notrace :-
  '$trace_set'(no_trace).
```

There are two convenience predicates for setting the trace mode, trace/0 and notrace/0.

These can be used to start and stop tracing of a query, such as:

```
?-  trace, mem(X, [a,b]),
    mem(X, [c,b]), notrace.
```

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

This query traces the evaluation of the two mem/2 goals then stops the trace.

The call/1 predicate.

```
call(Goal):-
  term_variables(Goal, Vars),
  compile_clause_anon(
    query(Vars):-Goal),
  !,
  '$jmp'(Vars).
```

The details of the call/1 predicate for a Prolog implementation are beyond the scope of the WAM design. We use a sketch of an implementation of call/1.

The compile_clause_anon/1 goal compiles the Vars and Goal into a predicate, but does not actually declare it anywhere. The functor is therefore irrelevant. The builtin '$jmp'/1 predicate calls the anonymous predicate.

The '$jmp'/1 builtin uses a *jmp* procedure.

```
procedure jmp(vars);
  begin
    if (TC = 3) then TC <- 2;
    {set P to address of newly
compiled code for query of Goal}
  end;
```

The TC register is conditionally advanced from mode 3 to 2 (*trace_next_jmp* to *trace_next*). Using the *trace_next_jmp* mode allows for calls to occur in the implementation of call/1 where those calls do not advance the mode from *trace_next* to *trace* and are not themselves traced.

WAM Modification Summary

The WAM instructions modified in the following text are: call, exec, try_me_else, retry_me_else, trust_me, try, retry, and trust. The modified procedure is *backtrack*. New procedures are: *setup_trace_call*, *adv_next_trace_cond*, *comp_call_or_execute*, and *backtrack_trace*.

The setup_trace_call procedure.

The setup_trace_call procedure is used in the call and execute instructions to prepare the WAM to execute '$trace'/3 predicate.

This procedure sets up the values in the A1 and A2 registers. The A1 registers either holds an atom when the procedure being traced has no arguments (arity is 0) or it holds a structure when the procedure being traced has 1 or more arguments (arity >= 1). (The *fun*(***P***) function returns the functor of the ***P*** predicate. The *ar*(***P***) function returns the arity of the ***P*** predicate.)

The statement '**put_structure P**,X(**N**+1);' creates a structure value in register $X_{(N+1)}$ which is guaranteed to be an unused register when evaluating this **call** instruction since there are only N permanent registers in use when this **call** is evaluated.

```
procedure setup_trace_call(
    P:predicate):
  begin
    if(arity(P) = 0)
      then
        put_value functor(P),A1;
      else
        begin
          put_structure
            fun(P)/ar(P),X(N+1);
          for i = 1 to ar(P)
            begin
              set_value Ai;
            end;
          put_value X(N+1),A1;
        end;
  end;
```

The adv_next_trace_cond procedure

This procedure advances the mode to 'trace' conditional on if it is currently 'trace_next', otherwise it leaves the mode unchanged.

```
procedure adv_next_trace_cond
    begin
        if (TC = 2) then TC <- 1;
    end;
```

The comp_call_or_execute procedure

The *comp_call_or_execute* procedure implements common steps used in both the call and execute instructions to complete their processing including setting the B0 register and the P register. This procedure handles the check for tracing and sets up for calling tracing if appropriate.

```
procedure comp_call_or_execute
    (P:predicate):
  begin
    B0 <- B;
    if (TC = 1 && traceable(P))
      then
        begin
          TC <- 0; // no trace
          setup_trace_call;
          num_of_args = 1;
          P <- TP;
        end
      else
        begin
          adv_next_trace_cond;
          num_of_args <- ar(P);
          P <- @(P);
        end;
  end;
```

The call instruction

Original version

The final WAM call instruction (p. 106 in [Aït-Kaci, 1999]) is **call *P, N*** where ***P*** is a predicate *p/n* and ***N*** is the number of stack variables *remaining in the current environment*.

[The ***N*** parameter is not used in the implementation of the **call** instruction - it is used by the **allocate** instruction when determining how many permanent variable slots are needed in the Environment when evaluating the code addressed by the **call** instruction.]

Traceable

This implementation must be modified to check if the current trace_call mode (TC = 1, *trace*) requires tracing and that the predicate ***P*** is traceable. The predicates with names starting with '$trace' and the true/0 and fail/0 predicates are not traceable. These predicates are protected from tracing to prevent unbounded recursion when the tracing mechanism attempts to trace itself.

Debug version

If tracing is required and appropriate then the predicate ***P*** and arguments A_i are copied to a HEAP structure using **put_structure** and **set_value** by the *setup_trace_call* procedure and the **call** instruction sets up to call the '$trace'/1 predicate by setting the *num_of_args* to 1 and the next code instruction address to TP.

From the implementation of the execute instruction developed below the 'completion' of the instruction after the setting of the CP register is the same in both the call and execute instructions. This is the *comp_call_or_execute* procedure.

The collapsed version rewrites the call instruction using the *comp_call_or_execute* procedure.

The execute instruction

The **execute** instruction (p. 107 in [Aït-Kaci, 1999]) is a simplified version of the **call** instruction.

```
Original version:

if defined(P)
  then
    begin
      CP <- P + inst_size(P);
      num_of_args <- ar(P);
      B0 <- B;
      P <- @(P)
  else backtrack;


function traceable
    (P:predicate): boolean
  return
    !(fun(P) startsWith '$trace'
      || P = true/0
      || P = fail/0);


Debug version:

expanded:
if defined(P)
  then
    begin
      CP <- P + inst_size(P);
      B0 <- B;
      if (TC = 1 && traceable(P))
        then
          begin
            TC <- 0; // no trace
            setup_trace_call;
            num_of_args = 1;
            P <- TP;
          end
        else
          begin
            adv_next_trace_cond;
            num_of_args <- ar(P);
            P <- @(P);
          end
    end
  else backtrack;

Debug version collapsed:
if defined(P)
  then
    begin
      CP <- P + inst_size(P);
      comp_call_or_execute(P);
    end
  else backtrack;
```

execute P: where **P** is the predicate to be evaluated.

This code is the same as for **call** but without the '`CP <- P + inst_size(P);`' statement.

Debug version

The version of this instruction extended for tracing is similar to the extended **call** instruction.

Using the *comp_call_or_execute* procedure that contains the steps common between the call and execute instructions the implementation of the execute instruction can be restated as shown.

```
Original version:

if defined(P)
  then
    begin
      num_of_args <- arity(P);
      B0 <- B;
      P <- @(P)
    end
  else backtrack;


Debug version expanded:

if defined(P)
  then
    begin
      B0 <- B;
      if (TC = 1 && traceable(P))
        then
          begin
            TC <- 0; // no trace
            setup_trace_call;
            num_of_args = 1;
            P <- TP;
          end
        else
          begin
            num_of_args <- ar(P);
            P <- @(P);
          end;

    end
  else backtrack;

Debug version collapsed:

if defined(P)
    then comp_call_or_execute;
    else backtrack;
```

The try_me_else instruction

The **try_me_else** instruction (p. 108 in [Aït-Kaci, 1999]) sets up a new choice-point frame. This frame is extended with one new slot for the TC ('trace call') register.

try_me_else L: where **L** is the code area address of the instructions for the next clause. The backtrack function sets P to **L** (from slot n+4 of the choice-point frame).

The frame size is increased to 9 and there is a new slot for TC.

```
Original:

if E > B
  then
    newB <-
      E +
      CODE[STACK[E+1] - 1] +
      2;
  else
    newB <-
      B +
      STACK[B] +
      8;
STACK[newB] <- num_of_args;
n <- STACK[newB];
for i <- 1 to n
  do STACK[newB + i] <- Ai;
STACK[newB + n + 1] <- E;
STACK[newB + n + 2] <- CP;
STACK[newB + n + 3] <- B;
STACK[newB + n + 4] <- L;
STACK[newB + n + 5] <- TR;
STACK[newB + n + 6] <- H;
STACK[newB + n + 7] <- B0;
B <- newB;
HB <- H;
P <- P + inst_size(P);

Debug:

if E > B
    then
      newB <-
        E +
        CODE[STACK[E+1] - 1] +
        2;
    else newB <-
       B + STACK[B] + 9;
STACK[newB] <- num_of_args;
n <- STACK[newB];
for i <- 1 to n
  do STACK[newB + i] <- Ai;
STACK[newB + n + 1] <- E;
STACK[newB + n + 2] <- CP;
STACK[newB + n + 3] <- B;
STACK[newB + n + 4] <- L;
STACK[newB + n + 5] <- TR;
STACK[newB + n + 6] <- H;
STACK[newB + n + 7] <- B0;
STACK[newB + n + 8] <- TC;
B <- newB;
HB <- H;
P <- P + inst_size(P);
```

The backtrack_trace procedure

The backtrack_trace procedure is common to several instructions. At this point it is a simple single assignment. It provides a single place to handle changes to these instructions across the different versions of tracing mechanisms.

```
procedure backtrack_trace(
    B:integer, n:integer):
  begin
      TC <- STACK[B + n + 8];
  end;
```

The retry_me_else instruction

retry_me_else L: After backtracking to the current choice-point, reset registers, update next clause to **L**, and continue with the next instruction.

The debug version is nearly identical with the original version. It only adds the resetting of TC.

```
original:

n <- STACK[B];
for i <- 1 to n
  do Ai <- STACK[B+i];
E <- STACK[B + n + 1];
CP <- STACK[B + n + 2];
STACK[B + n + 4] <- L;
unwind_trail(
  STACK[B + n + 5], TR);
TR <- STACK[B + n + 5];
HB <- H;
P <- P + inst_size(P);

debug:

n <- STACK[B];
for i <- 1 to n
  do Ai <- STACK[B+i];
E <- STACK[B + n + 1];
CP <- STACK[B + n + 2];
STACK[B + n + 4] <- L;
unwind_trail(
  STACK[B + n + 5], TR);
TR <- STACK[B + n + 5];
HB <- H;
backtrack_trace(B, n);
P <- P + inst_size(P);
```

The trust_me instruction

trust_me: After backtracking to the current choice-point, reset registers, discard the choice-point, and continue with the next instruction.

Note that the statement setting HB depends on the value of B and that the value of B is set in the immediately previous statement.

The trace version extends the original version to reset the TC register using the *backtrack_trace* procedure.

```
original:

n <- STACK[B];
for i <- 1 to n
  do Ai <- STACK[B + i];
E <- STACK[B + n + 1];
CP <- STACK[B + n + 2];
unwind_trail(
  STACK[B + n + 5], TR);
TR <- STACK[B + n + 5];
H <- STACK[B + n + 6];
B <- STACK[B + n + 3];

HB <- STACK[B + n + 6];
P <- P + inst_size(P);


debug:

n <- STACK[B];
for i <- 1 to n
  do Ai <- STACK[B + i];
E <- STACK[B + n + 1];
CP <- STACK[B + n + 2];
unwind_trail(
  STACK[B + n + 5], TR);
TR <- STACK[B + n + 5];
H <- STACK[B + n + 6];
backtrack_trace(B, n);
B <- STACK[B + n + 3];

HB <- STACK[B + n + 6];
P <- P + inst_size(P);
```

The try instruction

try L: Set up new choice-point frame and continue execution to code address **L**.

The frame size is increased to 9 and there is a new slot for TC.

```
original:

if E > B
  then
    newB <- E +
      CODE[STACK[E+1] - 1] +
      2;
    else newB <- B +
      STACK[B] +
      8;
STACK[newB] <- num_of_args;
n <- STACK[newB];
for i <- 1 to n
  do STACK[newB + i] <- Ai;
STACK[newB + n + 1] <- E;
STACK[newB + n + 2] <- CP;
STACK[newB + n + 3] <- B;
STACK[newB + n + 4] <-
  P + inst_size(P);
STACK[newB + n + 5] <- TR;
STACK[newB + n + 6] <- H;
STACK[newB + n + 7] <- B0;
B <- newB;
HB <- H;
P <- L;

debug:

if E > B
  then newB <-
    E +
    CODE[STACK[E+1] - 1] +
    2;
  else newB <-
    B +
    STACK[B] +
    9;
STACK[newB] <- num_of_args;
n <- STACK[newB];
for i <- 1 to n
  do STACK[newB + i] <- Ai;
STACK[newB + n + 1] <- E;
STACK[newB + n + 2] <- CP;
STACK[newB + n + 3] <- B;
STACK[newB + n + 4] <-
  P + inst_size(P);
STACK[newB + n + 5] <- TR;
STACK[newB + n + 6] <- H;
STACK[newB + n + 7] <- B0;
STACK[newB + n + 8] <- TC;
B <- newB;
HB <- H;
P <- L;
```

The retry instruction

retry L: After backtracking to the current choicepoint, reset registers, update next clause to the next instruction, and continue with **L**.

The trace version is nearly identical with the original version. It only adds the resetting of TC.

```
original:

n <- STACK[B];
for i <- 1 to n
  do Ai <- STACK[B+i];
E <- STACK[B + n + 1];
CP <- STACK[B + n + 2];
STACK[B + n + 4] <-
  P + inst_size(P);
unwind_trail(
  STACK[B + n + 5], TR);
TR <- STACK[B + n + 5];
H <- STACK[B + n + 6];
HB <- H;
P <- L;

debug:

n <- STACK[B];
for i <- 1 to n
  do Ai <- STACK[B+i];
E <- STACK[B + n + 1];
CP <- STACK[B + n + 2];
STACK[B + n + 4] <-
  P + inst_size(P);
unwind_trail(
  STACK[B + n + 5], TR);
TR <- STACK[B + n + 5];
H <- STACK[B + n + 6];
HB <- H;
backtrack_trace(B, n);
P <- L;
```

The trust instruction

trust L: After backtracking to the current choice-point, reset registers, discard the choice-point, and continue with **L**.

Note that the statement setting HB depends on the value of B and that the value of B is set in the immediately previous statement.

The trace version extends the original version to reset the TC register using the *backtrack_trace* procedure.

The backtrack procedure

The tutorial WAM uses a *backtrack* procedure to reset the B0 and P registers. To the right are the original and trace versions of this procedure.

In the trace version the TC register is restored when backtracking in the call to *backtrace_trace*.

```
original:

n <- STACK[B];
for i <- 1 to n
  do Ai <- STACK[B + i];
E <- STACK[B + n + 1];
CP <- STACK[B + n + 2];
unwind_trail(
  STACK[B + n + 5], TR);
TR <- STACK[B + n + 5];
H <- STACK[B + n + 6];
B <- STACK[B + n + 3];
HB <- STACK[B + n + 6];
P <- L;

debug:

n <- STACK[B];
for i <- 1 to n
  do Ai <- STACK[B + i];
E <- STACK[B + n + 1];
CP <- STACK[B + n + 2];
unwind_trail(
  STACK[B + n + 5], TR);
TR <- STACK[B + n + 5];
H <- STACK[B + n + 6];
backtrack_trace(B, n);
B <- STACK[B + n + 3];
HB <- STACK[B + n + 6];
P <- L;
```

```
procedure backtrack; // original
  begin
    if B = bottom_of_stack
      then fail_and_exit_program
      else
        begin
          B0 <- STACK[
              B + STACK[B] + 7];
          P <- STACK[
              B + STACK[B] + 4];
        end;
    end backtrack;

procedure backtrack; // trace
  begin
    if B = bottom_of_stack
      then fail_and_exit_program
      else
        begin
          B0 <- STACK[
              B + STACK[B] + 7];
          backtrack_trace(B, n);
          P <- STACK[
             B + STACK[B] + 4];
        end;
  end backtrack;
```

Extending the trace mechanism to display depth

This section develops the trace mechanism analogous to the interpret3/1 example. This version requires that the WAM and the '$trace'/2 predicate (extending the '$trace'/1 predicate) manage the Ancestors list. The Ancestors list is stored on HEAP and the address of the list is in the TI register. In most respects this version is the same as above.

The major changes are in the '$trace'/2 predicate and in the call and execute instructions.

The changes to the call and execute instructions are in the common *comp_call_or_execute* function. All of the instructions that manage the TC register must also manage the TI register. The call/1 predicate is unchanged.

The '$trace'/2 predicate.

The '$trace'/2 predicate that produces the tracing with depth numbers is analogous to the interpret3/2 predicate discussed above.

The '$trace'/2 changes include adding the `'$trace_push_info'(Goal, Anc)` goal to extend the *Anc* (Ancestors) list with *Goal*. '$trace_push_info'/2 uses the new '$trace_set_info'/1 builtin.

The '$trace_msg'/3 predicate is extended to '$trace_msg'/4 with the *Anc* parameter.

The '$trace_set_info'/1 builtin predicate sets the WAM TI ("trace info") register directly using the *trace_set_info* procedure.

WAM Modification Summary

The modified instructions are **try_me_else** and **try**. The modified procedures are *setup_trace_call*, *trace_call_or_execute*, *suspend_trace*, *comp_call_or_execute*, and *backup_trace*.

```
'$trace'(notrace, _) :-
  !,
  '$trace_set'(no_trace).

'$trace'(Goal, Anc) :-
  '$trace_set'(no_trace),
  '$trace_msg'('Call', 'Fail',
    Goal, Anc),
  '$trace_push_info'(Goal, Anc),
  '$trace_set'(trace_next_jmp),
  call(Goal),
  '$trace_set'(no_trace),
  '$trace_msg'('Exit', 'Redo',
    Goal, Anc),
  '$trace_set'(trace).

'$trace_push_info'(Goal, Anc) :-
  format(atom(X), '~w\n',
    [Goal]),
  '$trace_set_info'([X|Anc]).
'$trace_push_info'(_, Anc) :-
  '$trace_set_info'(Anc),
  !, fail.

'$trace_msg'(Success, _, Goal,
    Anc) :-
  '$trace_msg1'(Success, Goal,
    Anc).

'$trace_msg'(_, Failure, Goal,
    Anc) :-
  '$trace_msg1'(Failure, Goal,
    Anc),
  !,
  fail.


'$trace_msg1'(Label, Goal,
    Anc) :-
  length(Ancestors, K),
  write(K),
  write(' '),
  write(Label),
  write(' '),
  writeln(Goal).
```

```
procedure trace_set_info(
    info:integer);
  begin
    TI <- info;
  end;
```

The setup_trace_call procedure.

The setup_trace_call procedure is used in the call and execute instructions to prepare the WAM to execute '$trace'/3 predicate.

This procedure sets up the values in the A1 and A2 registers. The A1 register either holds an atom when the procedure being traced has no arguments (arity is 0) or it holds a structure when the procedure being traces has 1 or more arguments (arity >= 1). The A2 register is the trace info - the Ancestors list.

```
procedure setup_trace_call(
    P:predicate):
  begin
    if(ar(P) = 0)
      then
        put_value functor(P),A1;
      else
        begin
          put_structure
            fun(P)/ar(P),X(N+1);
          for i = 1 to ar(P)
             begin
               set_value Ai;
             end;
           put_value X(N+1),A1;
         end;
         put_value TI,A2;
    end;
```

The trace_call_or_execute function

The *trace_call_or_execute* function returns true if the input predicate ***P*** is traceable (e.g. not a special predicate such as '$trace' or 'true') and if the TC register is 1.

```
procedure trace_call_or_execute(
    P:predicate) returns boolean
  begin
    return
      TC = 1 && traceable(P);
    end;
```

The suspend_trace procedure

The *suspend_trace* function changes the trace mode to that mode's suspended version: trace -> no_trace.

```
procedure suspend_trace
  begin
    if (TC = 1) then TC <- 0;
  end;
```

The comp_call_or_execute procedure

The *comp_call_or_execute* procedure implementation is changed to expect 2 arguments for '$trace'/2 as set up by the new version of the *setup_trace_call* function.

```
procedure comp_call_or_execute(
    P:predicate):
  begin
    B0 <- B;
    if (trace_call_or_execute(P))
      then
        begin
          suspend_trace;
          setup_trace_call(P);
          num_of_args = 2;
          P <- TP;
        end
      else
        begin
          adv_next_trace_cond;
          num_of_args <- ar(P);
          P <- @(P);
        end;
```

The call instruction

The call instruction for depth-extended tracing adds management of the 'Ancestors' argument. The changes for this management are in the *comp_call_or_execute* and *setup_trace_call* functions.

The $trace/2 predicate has 2 arguments instead of 1. The second argument is the ancestors list set by the $trace/2 predicate before evaluating the call/1 goal.

```
if defined(P)
  then
    begin
      CP <- P + inst_size(P);
      comp_call_or_execute(P);
    end
  else backtrack;
```

The execute instruction

The version of this instruction extended for tracing is similar to the extended **call** instruction, relying on changes in the *comp_call_or_execute* and *setup_trace_call* functions.

This code is the same as for **call** but without the 'CP <- P + inst_size(P);' statement.

```
if defined(P)
    then comp_call_or_execute(P);
    else backtrack;
```

The backtrack_trace procedure

The backtrack_trace procedure is extended to restore the TI register. This handles the only change needed to support the retry_me_else, trust_me, retry, and trust instructions. Also this change adapts the backtrack procedure.

```
procedure backtrack_trace(
    B:integer, n:integer):
  begin
    TC <- STACK[B + n + 8];
    TI <- STACK[B + n + 9];
  end;
```

The try_me_else instruction

The frame size is increased to 10 and there is a new slot for TI.

```
if E > B
  then newB <-
    E + CODE[STACK[E+1] - 1] + 2
  else newB <- B + STACK[B] + 10;
STACK[newB] <- num_of_args;
n <- STACK[newB];
for i <- 1 to n
  do STACK[newB + i] <- Ai;
STACK[newB + n + 1] <- E;
STACK[newB + n + 2] <- CP;
STACK[newB + n + 3] <- B;
STACK[newB + n + 4] <- L;
STACK[newB + n + 5] <- TR;
STACK[newB + n + 6] <- H;
STACK[newB + n + 7] <- B0;
STACK[newB + n + 8] <- TC;
STACK[newB + n + 9] <- TI;
B <- newB;
HB <- H;
P <- P + inst_size(P);
```

The try instruction

The frame size is increased to 10 and there is a new slot for TI.

```
if E > B
  then newB <- E +
CODE[STACK[E+1] - 1] + 2
  else newB <- B + STACK[B] + 10;
STACK[newB] <- num_of_args;
n <- STACK[newB];
for i <- 1 to n
  do STACK[newB + i] <- Ai;
STACK[newB + n + 1] <- E;
STACK[newB + n + 2] <- CP;
STACK[newB + n + 3] <- B;
STACK[newB + n + 4] <- P +
instruction_size(P);
STACK[newB + n + 5] <- TR;
STACK[newB + n + 6] <- H;
STACK[newB + n + 7] <- B0;
STACK[newB + n + 8] <- TC;
STACK[newB + n + 9] <- TI;
B <- newB;
HB <- H;
P <- L;
```

Extending the trace mechanism to identify each procedure invocation

This section develops the trace mechanism analogous to the interpret4/1 example. This version requires that the WAM and the '$trace'/3 predicate (extending the '$trace'/2 predicate) handle the trace identifier. The trace identifier list is stored in the TD register. In most respects this version is the same as above. The only changes are in the '$trace'/3 predicate and in the call and execute instructions. The try_me_else, retry_me_else, trust_me, try, retry, and trust instructions are all the same as for the previous version. The *backtrack* function is also unchanged.

The '$trace'/3 predicate.

The '$trace'/3 predicate that produces the tracing with depth numbers is analogous to the interpret4/2 predicate discussed above.

The '$trace_push_info'/2 predicate is changed to '$trace_push_info'/3 with the addition of the *ID* parameter. The Ancestors list is now a list of pairs of *ID* and *Goal*-as-string.

WAM Modification Summary

The modified instructions are **call** and **execute**. The modified procedures are *setup_trace_call* and *comp_call_or_execute.*

The setup_trace_call procedure.

The setup_trace_call function is used in the call and execute instructions to prepare the WAM to execute '$trace'/3 predicate.

This procedure sets up the values in the A1, A2, and A3 registers. The A1 registers either holds an atom when the procedure being traced has no arguments (arity is 0) or it holds a structure when the procedure being traces has 1 or more arguments (arity >= 1). The A2 register is the trace info

```
'$trace'(notrace, _) :-
    !,
    '$trace_set'(no_trace).

'$trace'(Goal, Anc, ID) :-
  '$trace_set'(no_trace),
  '$trace_msg'('Call', 'Fail',
    Goal, Anc, ID),
  '$trace_push_info'(
    ID, Goal, Anc),
  '$trace_set'(trace_next_jmp),
  call(Goal),
  '$trace_set'(no_trace),
  '$trace_msg'('Exit', 'Redo',
    Goal, Anc, ID),
  '$trace_set'(trace).

'$trace_push_info'(
    ID, Goal, Anc) :-
  format(atom(X), '~w\n',
    [Goal]),
  '$trace_set_info'([ID-X|Anc]).
'$trace_push_info'(_, _, Anc) :-
  '$trace_set_info'(Anc),
  !, fail.

'$trace_msg'(Success, _, Goal,
    Anc, ID) :-
  '$trace_msg1'(Success, Goal,
    Anc, ID).

'$trace_msg'(_, Failure, Goal,
    Anc, ID) :-
  '$trace_msg1'(Failure, Goal,
    Anc, ID), !, fail.
```

```
procedure setup_trace_call:
  begin
    if(ar(P) = 0)
    then
      put_value fun(P),A1;
    else
      begin
        put_structure
          fun(P)/ar(P),X(N+1);
        for i = 1 to arity(P)
          begin
             set_value Ai;
          end;
          put_value X(N+1),A1;
      end;
    put_value TI,A2;
    put_value TD,A3;
  end;
```

- the Ancestors list. The A3 register is the incremented invocation identifier.

The comp_call_or_execute procedure

The *comp_call_or_execute* procedure implementation is changed to increment the TD register and to expect 3 arguments for '$trace'/3 as set up by the new version of the *setup_trace_call* function.

```
procedure comp_call_or_execute(
    P:predicate):
  begin
    B0 <- B;
    if (trace_call_or_execute(P))
      then
        begin
          suspend_trace();
          TD <- TD + 1;
          setup_trace_call(P);
          num_of_args = 3;
          P <- TP;
        end
      else
        begin
          adv_next_trace_cond;
          num_of_args <- ar(P);
          P <- @(P);
        end;
  end;
```

The call instruction

The call instruction for depth-extended tracing adds handling of the invocation identifier argument. These changes are handled in the *setup_trace_call* and *comp_call_or_execute* functions.

The $trace/3 predicate has 3 arguments instead of 2. The third argument is the invocation identifier incremented in the WAM just prior to an invocation of '$trace'/3.

```
if defined(P)
  then
    begin
      CP <- P + inst_size(P);
      comp_call_or_execute(P);
    end
  else backtrack;
```

The execute instruction

The version of this instruction extended for tracing is similar to the extended **call** instruction.

This code is the same as for **call** but without the '`CP <- P + instruction_size(P);`' statement.

```
if defined(P)
    then comp_call_or_execute(P);
    else backtrack;
```

Controlling the Trace

The trace produced by the WAM at this point is complete for the four ports, **Call**, **Exit**, **Fail**, and **Redo**. It provides the same information as the interpret4/1 example predicate. There are several enhancements that make this implementation much more useful. One enhancement is to allow the user to interact with the trace and *skip, creep, retry, leap, fail, abort,* or *disable* the trace at each port of each invocation. A second enhancement is to allow the user to specify predicate invocations on which to *spy* (i.e. setting break points) and the *port* at which to interact. A third enhancement is to allow the user to explore the current execution environment: *write* or *print* values, show *ancestors*, show *alternatives*, or show *listing* of the current predicate.

interpret5/1: interactive control of tracing.

The interpret5/1 predicate implementation extends interpret4/1 to support interactive control of the trace. This implementation includes control commands to *skip, creep,* and *fail.* The *ancestors* command displays the goal call stack of the current goal.

The interpret5/4 predicate includes two arguments to handle the interactive choice for the tracing mode, ‘trace’ or ‘notrace’. This mode is input to trace5/5: ‘notrace’ skips the interaction and the goal trace output, ‘trace’ invokes the interaction with the user. The interpret5/4 predicate is written using the DCG notation (e.g. ‘interpret5(true, _) --> !.’ is expanded to ‘interpret5(true, _, Mode, Mode) :- !.’).

```
interpret5(Goal) :-
  clear_invocation,
  interpret5(Goal, [], trace, _).


interpret5(true, _) --> !.

interpret5((G1, G2), Anc) -->
  !,
  interpret5(G1, Anc),
  interpret5(G2, Anc).

interpret5(Goal, Anc) -->
  {increment_invocation(K)},
  trace5(Goal, Anc, K).
```

The trace5/5 predicate recursively invokes the interpret5/4 predicate for the body goals in a clause of Goal.

```
trace5(Goal, Anc, K, notrace, notrace) :-
  cls(Goal, Body),
  interpret5(Body,[Goal| Anc], notrace, notrace).
trace5(Goal, Anc, K, trace, NextMode) :-
  NewAnc = [Goal| Anc],
  interact(call, fail, NewAnc, K, InterimMode),
  cls(Goal, MoreGoals),
  interpret5(MoreGoals, NewAnc, InterimMode, _),
  interact(exit, redo, NewAnc, K, NextMode).
```

It calls the interact/5 predicate to tell the user the current goal information, read the user's command (a single character), and handle that command.

```
interact(Success, _, Stack,
    Inv, NextMode) :-
  interact_port(Success, Stack, Inv, NextMode).

interact(_, Failure, Stack, Inv, _) :-
  interact_port(Failure, Stack, Inv, _), !, fail.


interact_port(Port, Stack,
    Inv, NextMode) :-
  prompt(Port, Stack, Inv),
  read_and_execute(Port, Stack,
    Inv, NextMode).


prompt(Port, [Goal|Anc], Inv) :-
  length(Anc, Depth),
  write_list([Inv, Depth, Port, ':', Goal, '?'],
    ' ').

read_and_execute(Port, Stack, Inv, NextMode) :-
  repeat, get_char(Command),
  check_command(Command),
  !,
  execute(Command, Port, Stack, Inv, NextMode).

check_command(Command) :-
  member(Command, [c, s, f, r, g])
  ;
  writeln('Commands are: "c" (creep), "s" (skip),
"f" (fail), "r" (retry), or "g" (ancestors).'),
  fail.

execute(c, _, _, _, trace) :- !.
execute(s, call, _, _, notrace) :- !.
execute(s, _, _, _, notrace) :- !.
execute(f, _, _, _, _) :- !, fail.
execute(g, Port, [Goal|Anc], Inv, Mode) :-
  write('Ancestors: '),
  writenl(Anc),
  read_and_execute(Port, [Goal|Anc], Inv, Mode).
```

‘$traceR’/3: enhanced WAM trace predicate with user interaction.

The enhanced ‘$traceR’/3 predicate is analogous to the trace5/5 predicate. This mechanism supports the creep, skip, fail, ancestors, retry and abort commands. The ‘$trace_retry’ predicate creates a choice-point and uses a special builtin ‘$get_backtrack_frame’/1 to get the address of the frame for that choice-point. This choice-point is passed through to the ‘$trace_cmd’/6 clause for ‘r’ (retry) where the body sets the backtrack frame register to the retry choice-point then fails. The failure makes the WAM reset the state to the choice-point frame and continue evaluation in that frame (unwinding stacks and freeing data as appropriate). The evaluation continuation uses the second clause of the invocation of the ‘$trace_retry’/1 predicate that created the choice-point. This restarts the evaluation of Goal. The user interaction depends on a read_char/1 predicate that reads a command character. The implementation of this predicate depends on the details of the Prolog system.

```
'$traceR'(notrace, _, _) :-
  !,
  '$trace_set'(no_trace).

'$traceR'(Goal, Anc, ID) :-
  '$trace_set'(no_trace),
  '$trace_retry'(Bk),
  '$trace_interact'(call, fail, Goal, Anc, ID, Bk),
  call(Goal),
  '$trace_set'(no_trace),
  '$trace_set_info'(Anc),
  '$trace_interact'(exit, redo, Goal, Anc, ID, Bk).

'$trace_retry'(Bk) :- '$get_backtrack_frame'(Bk).
'$trace_retry'(Bk) :- '$trace_retry'(Bk).

'$trace_interact'(A, _B, G, Anc, ID, Bk) :-
  '$trace_interact'(A, G, Anc, ID, Bk).
'$trace_interact'(_A, B, G, Anc, ID, Bk) :-
  '$trace_interact'(B, G, Anc, ID, Bk), !, fail.

'$trace_interact'(P, G, Anc, ID, B) :-
  '$trace_prompt'(P, G, Anc, ID),
  '$trace_read_and_cmd'(P, G, Anc, ID, B).

'$trace_prompt'(Port, Goal, Ancestors, ID) :-
  '$trace_create_prompt'(K, Port, Goal, ID, Prompt),
  '$trace_set_prompt'(Prompt).

'$trace_create_prompt'(K, Goal, ID, Prompt) :-
  pad_number(ID, 7, PadID),
  pad_number(K, 5, PadK),
  concat_list([PadID, PadK, ' ', Goal], Prompt).
'$trace_create_prompt'(K, Port, Goal, ID, Prompt) :-
  pad_number(ID, 7, PadID),
  pad_number(K, 5, PadK),
  capitalize(Port, CapPort),
  concat_list([PadID, PadK, ' ',
    CapPort, ': ', Goal], Prompt).

'$trace_read_and_cmd'(P, G, Anc, ID, B) :-
  '$trace_check_command'(X),
  !,
  '$trace_cmd'(X, P, G, Anc, ID, B).

'$trace_check_command'(X) :-
  read_char(X),
  member(X, [c, s, f, r, g, a]),
  !.
'$trace_check_command'(X) :-
  writeln('Commands are: "c" (creep), "s" (skip),
"f" (fail), "r" (retry), "g" (ancestors),
“a” (abort).'),
  '$trace_check_command'(X).
```

The '$trace_cmd'/6 predicate interprets the debug command. It either displays information (the 'g' command), sets a trace flag (the 'c' and 's' commands), forces backtracking (the 'r' command), or halts the WAM engine (the 'a' command).

WAM Modification Summary

There are no changes required in the WAM to support the basic user interactions. (This assumes that the Prolog system based on the WAM has sufficient features to support the read_char/1 predicate.)

```
'$trace_cmd'(c, call, G, Anc, _, _) :-
  !,
  '$trace_push_info'(G, Anc),
  '$trace_set'(trace_next_jmp).
'$trace_cmd'(c, exit, _, _, _, _) :-
  !, '$trace_set'(trace).
'$trace_cmd'(c, _L, _, _, _, _) :- !.
'$trace_cmd'(s, call, _, _, _, _) :-
  !, '$trace_set'(no_trace).
'$trace_cmd'(s, _, _, _, _, _) :- !.
'$trace_cmd'(f, _, _, _, _, _) :- !, fail.
'$trace_cmd'(r, _, _, ID, _, Bk) :- % retry
  !,
  '$set_backtrack_frame'(Bk),
  fail.
'$trace_cmd'(a, _, _, _, _, _) :- !,halt.
'$trace_cmd'(g, P, G, Anc, ID, B) :- % ancestors
  !,
  write('Ancestors: '),
  '$trace_write_ancestors'(Anc),
  '$trace_read_and_cmd'(P, G, Anc, ID, B).

'$trace_write_ancestors'([]) :- !.
'$trace_write_ancestors'(Anc) :-
  reverse(Anc, RevAnc),
  writeln('Ancestors:'),
  '$trace_write_ancestors1'(RevAnc, 1).

'$trace_write_ancestors1'([],_).
'$trace_write_ancestors1'([ID-Goal|T], D) :-
  '$trace_create_prompt'(D, Goal, ID, Prompt),
  writeln(Prompt),
  DNext is D + 1,
  '$trace_write_ancestors1'(T, DNext).
```

'$traceR'/3: extending WAM trace predicate with 'nodebug' command.

This implementation is extended to support the 'nodebug' command. A new trace mode is introduced, 'skip_trace'. The WAM instructions that 'backtrack' the state.trace_call do *not* reset the value if it is already 'no_trace'. With this change backtracking to a frame that was tracing calls will not restore tracing if the 'nodebug' command has been used.

The changes to the $traceR predicate and supporting predicates are extensive. The '$trace_is_suspended'/0 and '$trace_suspend_if_active'/1 predicates are introduced to manage the skip_trace mode. (These predicates will be extended in the next version.)

```
'$traceR'(notrace, _, _) :-
  !,
  '$trace_set'(no_trace).

'$traceR'(Goal, Anc, ID) :-
  '$trace_set'(skip_trace),
  '$trace_retry'(ID, B),
  '$trace_interact'(call, fail, Goal, Anc, ID, B),
  call(Goal),
  '$trace_suspend_if_active',
  ('$trace_is_suspended'
    -> '$trace_set_info'(Anc),
       '$trace_interact'(exit, redo, Goal,
         Anc, ID, B)
  ; true).

'$trace_retry'(ID, B) :- '$get_backtrack_frame'(B).
'$trace_retry'(ID, B) :-
  '$trace_retry_value'(ID),
  '$trace_set_retry'(none),
  '$trace_retry'(ID, B).

'$trace_is_suspended' :-
  '$trace_value'(Value),
  '$trace_is_suspended'(Value),
  !.
'$trace_is_suspended'(skip_trace).

'$trace_suspend_if_active' :-
  '$trace_value'(Value),
  '$trace_suspend_if_active'(Value),
  !.
'$trace_suspend_if_active'(trace) :-
  !,
  '$trace_set'(skip_trace).
'$trace_suspend_if_active'(_).

'$trace_interact'(A, _B, G, Anc, ID, Bk) :-
  '$trace_interact'(A, G, Anc, ID, Bk).
'$trace_interact'(_A, B, G, Anc, ID, Bk) :-
  \+ '$trace_value'(no_trace),
  '$trace_interact'(B, G, Anc, ID, Bk),
  !,
  fail.

'$trace_check_command'(X) :-
  read_char(X),
  member(X, [c, s, f, r, g, a, n]),
  !.
'$trace_check_command'(X) :-
  writeln('Commands are: "c" (creep), "s" (skip),
"f" (fail), "r" (retry), "g" (ancestors),
“a” (abort), “n” (nodebug).'),
  '$trace_check_command'(X).
```

The major changes to the '$trace_cmd'/6 predicate are the implementations of the 's' (skip), 'n' (no trace) and 'r' (retry) commands. The skip command sets the trace mode to 'skip_trace' (instead of 'no_trace' in the previous version) and the no-trace command sets the mode to 'no_trace'.

In the new version of the retry command the invocation ID is recorded in the WAM state as the retry ID. This recorded retry ID is used by the '$trace_retry'/2 predicate to determine when the current invocation is the target of a retry backtrack.

WAM Modification Summary

No instructions are modified for the 'nodebug' feature. There are two new procedures: *get_backtrack_frame* and *set_backtrack_frame*. The changed procedures are: *trace_set*, *trace_value*, *backtrack_trace*, *suspend_trace*, and *comp_call_or_exec*.

```
'$trace_cmd'(c, call, G, Anc, ID, _) :-
    !,
    '$trace_push_info'(ID, G, Anc),
    '$trace_set'(trace_next_jmp).
'$trace_cmd'(c, exit, _, _, _, _) :-
  !, '$trace_set'(trace).
'$trace_cmd'(c, _L, _, _, _, _) :- !.
'$trace_cmd'(s, call, _, _, _, _) :-
  !, '$trace_set'(skip_trace).
'$trace_cmd'(s, _, _, _, _, _) :- !.
'$trace_cmd'(n, _, _, _, _, _) :- % nodebug
  !, '$trace_set'(no_trace).
'$trace_cmd'(f, _, _, _, _, _) :- !, fail.
'$trace_cmd'(r, _, _, _, ID, B) :- % retry
  !,
  '$trace_set_retry'(ID),
  '$set_backtrack_frame'(B),
  fail.
'$trace_cmd'(a, _, _, _, _, _) :- !,halt.
'$trace_cmd'(g, P, G, Anc, ID, B) :- % ancestors
  !,
  write('Ancestors: '),
  '$trace_write_ancestors'(Anc),
  '$trace_read_and_cmd'(P, G, Anc, ID, B).
```

The backtrack frame procedures

There are two new builtins that use two new procedures to access the B register for the current backtrack frame: *get_backtrack_frame* and *set_backtrack_frame*.

```
procedure get_backtrack_frame(BTerm:integer)
    returns boolean:
  begin
    return
      unify(BTerm, PL_integer_term(B));
    end;

procedure set_backtrack_frame(BTerm:integer)
    returns boolean:
  begin
    B = PL_integer(BTerm);
    return true;
  end;
```

The trace_set and trace_value builtins

The instructions are changed to handle the new skip_trace trace_call mode. The TC register uses value 4 for skip_trace.

```
procedure trace_set(modeTerm:integer)
    returns boolean;
  begin
    mode = PL_atom_chars(modeTerm);
    if (mode = 'no_trace') then TC <- 0
    else if (mode = 'trace') then TC <- 1
    else if (mode = 'trace_next') then TC <- 2
    else if (mode = 'trace_next_jmp') then TC <- 3
    else if (mode = 'skip_trace') then TC <- 4
    else ERROR;
    return true;
  end;

procedure trace_value(modeTerm:integer)
    returns boolean;
  begin
    if (TC = 0) then mode <- 'no_trace'
    else if (TC = 1) then mode <- 'trace'
    else if (TC = 2 ) then mode <- 'trace_next'
    else if (TC = 3) then mode <- 'trace_next_jmp'
    else if (TC = 4) then mode <- 'skip_trace'
    else ERROR;
    return unify(modeTerm, PL_atom_lookup(mode));
  end;
```

The backtrack_trace procedure

The *backtrack_trace* procedure is extended to not restore trace information if the TC register is set to 0 (no_trace). This handles the only change needed to support the retry_me_else, trust_me, retry, and trust instructions. Also this change adapts the backtrack procedure.

```
procedure backtrack_trace(B, n):
    begin
        if(TC != 0)
            begin
                TC <- STACK[B + n + 8];
                TI <- STACK[B + n + 9];
            end;
    end;
```

The suspend_trace function for skip_trace

```
procedure suspend_trace()
    begin
        if (TC = 1)
            then TC <- 4;
    end;
```

The *suspend_trace* function for skip_trace changes the trace mode to skip_trace (4) instead of no_trace (0).

The comp_call_or_execute function for skip_trace

The *comp_call_or_execute* function implementation is unchanged - it relies on the change to suspend_trace function to handle the change to skip_trace.

Spy points

The remaining basic debugging feature is a spy facility: the user specifies which predicates on which to spy. These are like break points in a conventional debugger. The leap command skips tracing until a spy predicate is encountered. There are new trace commands related to leaping and spying: “l” (leap), “+” (add a spypoint), and “-” (removed a spypoint).

Spy points are defined in different ways in different systems. In most systems there are both unconditional (or plain) spy points and conditional spy points. In this note we only address unconditional spy point. In GNU Prolog an unconditional spy point is defined for a collection of one or more predicates with the same functor. From p.34 of [Diaz, 2013]:

- Name: set a spy-point for any predicate whose name is Name (whatever the arity).
- Name/Arity: set a spy-point for the predicate whose name is Name and arity is Arity.
- Name/A1-A2: set a spy-point for the each predicate whose name is Name and arity is between A1 and A2.

SWI-Prolog has a similar definition for spy points, where the specification for a spy point is Name, Name/Arity, or Name//Arity. (The Name//Arity specification is the same as Name/Arity+2 and is convenient for specifying a predicate defined using DCG rules.) SWI-Prolog does not use the Name/A1-A2 form. SICStus Prolog supports Name and Name/Arity for specifying unconditional spy points (p. 233 in [Carlsson et al, 2019]). The XSB system supports plain spy point specifications of Name and Name/Arity (p. 327 in [Swift et al, 2013]).

In this note we use a single implementation for all spy points. A full spy point specification ‘spy(P, G, B)’ includes a port P, a goal G to match the current trace goal, and a prolog expression B. When the expression B is something other than ‘true’ then B and the match goal share one or more variables. Using the ‘+’ command when at a trace point for some goal G with functor F and arity A creates a spy point of ‘spy(_, F(_, _, ...), true)’. The ‘-’ command retracts all spy points for the functor and arity of the current goal: retractall(spy(_, F(_, _, ...), _)).

The spy(Name) goal creates spy facts for all known arities of predicates with functor Name. E.g. if p/0 and p/1 are both defined then spy(p) creates spy(_, p, true) and spy(_, p(_), true). The spy(Name/Arity) goal creates the spy fact for a single predicate. E.g. spy(p/1) creates spy(_, p(_), true).

There are new trace modes to support spy points and leaping: leap_trace, suspend_leap_trace, and leap_trace_next_jmp. The ‘leap’ mode cause $traceR to be invoked the same as ‘trace’ mode, but it causes tracing of ports and goals that are not spied to not be displayed, the same as ‘skip’ mode. The tracing mechanism is active when leaping for all goals to track the ancestors and to check each port of each goal for a spy point that is satisfied.

The ‘$traceR’ predicate is unchanged from above. Several support predicates are changed.

```
'$trace_is_suspended'(skip_trace).
'$trace_is_suspended'(suspend_leap_trace).

'$trace_suspend_if_active'(trace) :-
  !, '$trace_set'(skip_trace).
'$trace_suspend_if_active'(leap_trace) :-
  !, '$trace_set'(suspend_leap_trace).
'$trace_suspend_if_active'(_).

'$trace_interaction_enabled'(P, G) :-
  '$trace_spy_mode'(M),
  '$trace_interaction_enabled'(M, P, G).

'$trace_interaction_enabled'(all, _P, _G) :- !.
'$trace_interaction_enabled'(specified, P, G) :-
  !,
  % Following double-negative is used
  % to avoid persistent bindings
  % of variables in G (if any).
  \+ \+ (
    '$trace_spy_specification'(P, G, B),
    (B = true -> true ; call(B))
  ).

'$trace_spy_mode'(M) :-
    '$trace_value'(Value),
    '$trace_spy_mode1'(Value, M).

'$trace_spy_mode1'(trace, all).
'$trace_spy_mode1'(skip_trace, all).
'$trace_spy_mode1'(leap_trace, specified).
'$trace_spy_mode1'(suspend_leap_trace, specified).

'$trace_check_command'(X) :-
  read_char(X),
  member(X, [c, s, l, (+), (-), f, r, g, a, n]),
  !.

'$trace_check_command'(X) :-
  writeln('Commands are: "c" (creep), "s" (skip),
"l" (leap), "+" (spy this), "-" (nospy this),
"f" (fail), "r" (retry), "g" (ancestors),
"a" (abort), "n" (nodebug).'),
  '$trace_check_command'(X).
```

The '$trace_cmd'/6 predicate has the same clauses as previously plus the two additional clauses for '+' to add a spy point on a predicate and '-' to remove spy points for a predicate.

```
:- dynamic('$trace_spy_specification'/3).

(Other '$trace_cmd'/6 clauses as above)

'$trace_cmd'(+,  P, G, Anc, ID, B) :-
  !,
  G =.. [F|As], length(As, L),
  length(Ts, L), GT =.. [F|Ts],
  assertz('$trace_spy_specification'(_, GT, true)),
  write('Spypoint placed on '), writeln(F / L),
  '$trace_read_and_cmd'(P, G, Anc, ID, B).

'$trace_cmd'(-,  P, G, Anc, ID, B) :-
  !,
  G =.. [F|As], length(As, L),
  length(Ts, L), GT =.. [F|Ts],
  retractall('$trace_spy_specification'(_, GT, _)),
  write('Spypoint removed from '), writeln(F / L),
  '$trace_read_and_cmd'(P, G, Anc, ID, B).
```

WAM Modification Summary

No instructions are changed. Modified procedures are: *trace_get*, *trace_value*, *trace_call_or_exec*, *suspend_trace*, and *adv_next_trace_cond*.

The trace_set and trace_value builtins for leap_trace

The instructions are changed to handle the new *leap_trace* trace_call mode. The TC register uses value 5 for *leap_trace*, 6 for *suspend_leap_trace*, and 7 for *leap_trace_next*, and 8 for *leap_trace_next_jmp*.

```
procedure trace_set(modeTerm) returns boolean;
  begin
    mode = PL_atom_chars(modeTerm);
    if (mode = 'no_trace') then TC <- 0
    else if (mode = 'trace') then TC <- 1
    else if (mode = 'trace_next') then TC <- 2
    else if (mode = 'trace_next_jmp') then TC <- 3
    else if (mode = 'skip_trace') then TC <- 4
    else if (mode = 'leap_trace') then TC <- 5
    else if (mode = 'suspend_leap_trace')
      then TC <- 6
    else if (mode = 'leap_trace_next') then TC <- 7
    else if (mode = 'leap_trace_next_jmp')
      then TC <- 8
    else ERROR;
    return true;
  end;

procedure trace_value(modeTerm) returns boolean;
  begin
    if (TC = 0) then mode <- 'no_trace'
    else if (TC = 1) then mode <- 'trace'
    else if (TC = 2 ) then mode <- 'trace_next'
    else if (TC = 3) then mode <- 'trace_next_jmp'
    else if (TC = 4) then mode <- 'skip_trace'
    else if (TC = 5) then mode <- 'leap_trace'
    else if (TC = 6)
      then mode <- 'suspend_leap_trace'
    else if (TC = 7) then mode <- 'leap_trace_next'
    else if (TC = 8)
      then mode <- 'leap_trace_next_jmp'
    else ERROR;
    return unify(modeTerm, PL_atom_lookup(mode));
  end;
```

The trace_call_or_exec function for leap_trace

The *trace_or_call* function returns true if the input predicate ***P*** is traceable (e.g. not a special predicate such as '$trace' or 'true') and if the TC register is 1:

```
procedure trace_call_or_execute(
    P:predicate)
    returns boolean
  begin
    return
      (TC = 1 || TC = 5)
      && is_traceable(P);
    end;
```

The suspend_trace procedure for leap_trace

The *suspend_trace* procedure changes the trace mode to that mode's suspended version: trace -> no_trace.

```
procedure suspend_trace
    begin
        if (TC = 1)
            then TC <- 4
        else if (TC = 5)
            then TC <- 6
    end;
```

The adv_next_trace_cond procedure for leap_trace

This procedure changes the mode from *trace_next* to *trace* or *leap_trace_next* to *leap_trace*.

```
procedure adv_next_trace_cond
    begin
        if (TC = 2) then TC <- 1
        else if (TC = 7)
          then TC <- 5;
    end;
```

References

Aït-Kaci, 1999
Hassan Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction.* 1999. Reprinted from Cambridge, Mass: MIT Press edition. URL: http://wambook.sourceforge.net/wambook.pdf.

Cabeza et al, 2018
Daniel Cabeza, Manual C. Rodriguez, Edison Mera, A. Ciepielewski (first version), Mats Carlsson (first version), T. Chikayama (first version), K. Shen (first version). “Interactive debugger” in “The Ciao System: A New Generation, Multi-Paradigm Programming Language and Environment (Including a State-of-the-Art ISO-Prolog).” *Technical Report CLIP 3/97-1.18.* Version 1.18 (2018/12/6, 11:25:8 CEST). Edited by: Francisco Bueno, Manuel Carro, Manuel Hermenegildo, Pedro López, José F. Morales. The Computational logic, Languages, Implementation, and Parallelism (CLIP) Lab, School of CS, T. U. of Madrid (UPM), IMDEA Software Institute. https://web.archive.org/web/20190309190405/https://ciao-lang.org/ciao/build/doc/ciao.html/debugger_doc.html

Byrd, 1980
Lawrence Byrd. “Understanding the control flow of Prolog programs.” *Logic Programming Workshop*, 1980. (publisher unknown)

Carlsson et al, 2019
Mats Carlsson et al. SICStus Prolog User’s Manual. 2019. http://sicstus.sics.se/

Diaz, 2013
Daniel Diaz. *GNU Prolog: A Native Prolog Compiler with Constraint Solving over Finite Domains Edition 1.44, for GNU Prolog version 1.4.4*. April 23, 2013. archive.org URL: https://web.archive.org/web/20180516234545/http://gprolog.org/manual/gprolog.pdf.

Swift et al, 2013
Terrence Swift, David S. Warren, Konstantinos Sagonas, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, Luis de Castro, Rui F. Marques, Diptikalyan Saha, Steve Dawson, and Michael Kifer. *The XSB System, Version 3.3.x. Volume 1: Programmer’s Manual.* July 4, 2013. https://web.archive.org/web/20170531123709/http://xsb.sourceforge.net/downloads/manual1.pdf

Warren, 2018
David S. Warren. “WAM for everyone: a virtual machine for logic programming.” In *Declarative Logic Programming.* 2018. Michael Kifer and Yanhong Annie Liu (Eds.). Association for Computing Machinery and Morgan & Claypool, New York, NY, USA 237-277. DOI: https://doi.org/10.1145/3191315.3191320

Wielemaker, 2019

Jan Wielemaker. *SWI-Prolog Reference Manual* (Updated for version 8.0-2). March, 2019. URL: http://www.swi-prolog.org/download/stable/doc/SWI-Prolog-8.0.2.pdf