Appendix C

Formal specification of the Berkeley Abstract Machine semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Formal specification of the Berkeley Abstract Machine (BAM) semantics
% Copyright (C) 1990 Peter Van Roy and Regents of the University of California
% May be used and modified for non-commercial purposes if this notice is kept.
% Written by Peter Van Roy.

% The specification is a Prolog program that defines the meaning of BAM in
% terms of its execution in a simple memory model.  It runs BAM code directly
% from the output of the Aquarius compiler.

% The specification does not include the user instructions of the BAM since
% their behavior depends on the target machine.

% The specification is written in the Extended DCG notation.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Meaning of registers:
% r(b)      Index to most recent choice point.
% r(e)      Index to current environment.
% r(tr)     Top of trail stack.
% r(h)      Top of heap stack.
% r(hb)     Value of r(h) at last choice point creation.
% r(pc)     Code address.
% r(cp)     Continuation pointer for code.
% r(tmp_cp) Temporary continuation pointer for code, used only in simple_call.
% r(retry)  Retry address for backtracking, only exists inside choice points.
% r(I)      Argument and temporary register.
% p(I)      Location on current environment.

% Types stored in registers:
% r(e) Contains values of registers {r(e),r(cp)} U {p(0), ..., p(N-1)},
%      where N is the size of the environment.
% r(b) Contains values of registers {r(e),r(cp),r(tr),r(b),r(h),r(retry)} U RS,
%      where RS is a subset of {r(0), r(1), ...}.
% r(tr)         Contains a natural number.
% r(h), r(hb)   Contain words with a pointer tag.
% r(pc), r(cp)  Contain natural numbers or symbolic labels.
% r(tmp_cp)     Contains a symbolic label.
% r(retry)      Contains a symbolic label.
% r(I)          Contains a word.
% p(I)          Contains a word.

% Comments:
% A word is either an integer or a structure of the form Tag^Value where Value
% is a natural number except if Tag=tatm, in which case Value is an atom or a
% structure (F/N) where F is an atom and N is a natural number.
```

```
% A symbolic label is either the atom 'fail', or the structure F/N, or the
% structure l(F/N,I), where F is an atom and N and I are natural numbers.
% r(cp) is stored in environments, allowing nested calls.
% r(tmp_cp) is not stored in environments, allowing only one level of call.
% However, no environment is needed in a predicate containing a simple_call.
% There are no explicit stacks for environments or choice points; registers
% r(e) and r(b) each contain a set of register values.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Accumulator declarations:

% Accumulators:

acc_info(code,   T, In, Out, table_command(T,In,Out)).
acc_info(lblmap, T, In, Out, table_command(T,In,Out)).
acc_info(regs,   T, In, Out, table_command(T,In,Out)).
acc_info(trail,  T, In, Out, table_command(T,In,Out)).
acc_info(heap,   T, In, Out, heap_table_command(T,In,Out)).
acc_info(count,  T, In, Out, (Out is T+In)).

% Predicate declarations:

% Top level:
pred_info(           execute, 0, [regs,heap,trail,code,lblmap,count]).
pred_info(        instr_loop, 0, [regs,heap,trail,code,lblmap,count]).
pred_info(    instr_loop_end, 1, [regs,heap,trail,code,lblmap,count]).
pred_info(             instr, 1, [regs,heap,trail,code,lblmap]).
pred_info(        numeric_pc, 2, [lblmap]).

% Addressing modes:
pred_info(              heap, 3, [     heap]).
pred_info(               reg, 3, [regs     ]).
pred_info(              perm, 3, [regs     ]).
pred_info(             a_var, 3, [regs     ]).
pred_info(             var_i, 3, [regs,heap]).
pred_info(               arg, 2, [regs     ]).
pred_info(             arg_i, 2, [regs,heap]).
pred_info(            numreg, 2, [regs     ]).
pred_info(            numarg, 2, [regs,heap]).
pred_info(           var_off, 2, [regs,heap]).
pred_info(           imm_tag, 2, [regs     ]).
pred_info(              ea_e, 2, [regs,heap]).
pred_info(              ea_m, 2, [regs,heap]).
pred_info(              ea_p, 2, [regs,heap]).

% Instruction utilities:
pred_info(         deref_rtn, 2, [regs,heap,trail]).
pred_info(    deref_rtn_cont, 3, [regs,heap,trail]).
pred_info(         equal_rtn, 3, [regs,heap,trail]).
pred_info(        switch_rtn, 5, [regs,heap,trail]).
pred_info(          test_rtn, 4, [regs,heap,trail]).
pred_info(     jump_cond_rtn, 4, [regs,heap,trail]).
pred_info(       hash_lookup, 3, [regs,heap,trail,lblmap,code]).
```

```
pred_info(      hash_lookup_2, 3, [regs,heap,trail,lblmap,code]).
pred_info(      hash_indirect, 3, [     heap      ]).
pred_info(   save_choice_regs, 2, [regs,heap,trail]).
pred_info(restore_choice_regs, 2, [regs,heap,trail]).
pred_info(        detrail_rtn, 2, [regs,heap,trail]).
pred_info(          trail_rtn, 1, [regs,heap,trail]).
pred_info(          cmp_trail, 2, [regs,heap,trail]).
pred_info(          unify_rtn, 3, [regs,heap,trail]).
pred_info(        unify_rtn_2, 3, [regs,heap,trail]).
pred_info(        unify_rtn_2, 5, [regs,heap,trail]).
pred_info(   unify_rtn_args_2, 6, [regs,heap,trail]).
pred_info(   unify_rtn_args_3, 7, [regs,heap,trail]).
pred_info(          unify_atm, 3, [regs,heap,trail]).
pred_info(          unify_end, 2, [regs,heap,trail]).
pred_info(       unify_varvar, 2, [regs,heap,trail]).
pred_info(           get_size, 3, [     heap      ]).
pred_info(              arith, 4, [regs,heap      ]).

pred_info(          write_rtn, 0, [regs,heap,trail]).
pred_info(          write_rtn, 1, [regs,heap,trail]).
pred_info(          write_arg, 2, [regs,heap,trail]).
pred_info(         write_args, 3, [regs,heap,trail]).

% Implement the accumulator commands:
table_command(ins(I,Val), In,  In) :- ins(In, I, Val).
table_command(get(I,Val), In,  In) :- get(In, I, Val).
table_command(set(I,Val), In, Out) :- set(In, I, Val, Out).

% Mask off tag before looking up heap entry:
heap_table_command(ins(_^I,Val), In,  In) :- ins(In, I, Val).
heap_table_command(get(_^I,Val), In,  In) :- get(In, I, Val).
heap_table_command(set(_^I,Val), In, Out) :- set(In, I, Val, Out).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Initialization and runtime options ***

:- dynamic(bamspec_option/1).

main :-
   save(bamspec, 1),
   prompt(_, ''),
   ( copyright,
     execute
   ; error(['Sorry, the executable BAM specification has failed.'])
   ),
   halt.
main :-
   halt.

copyright :-
   write('Berkeley Abstract Machine (BAM) Executable Specification'), nl,
   write('Copyright (C) 1990 Peter Van Roy and '),
   write('Regents of the University of California'), nl, nl.
```

```
flag_print(I) :- bamspec_option(print), !, write('Executing '), write(I), nl.
flag_print(_).

% Look up symbolic label to get a numeric PC:
numeric_pc(PC,  PC) -->> {integer(PC)}, !.
numeric_pc(PC, NPC) -->> [get(PC,NPC)]:lblmap.

% Read in the instructions and create the code array and label map:
% The code array gives the instruction corresponding to each PC value.
% The label map gives the PC value corresponding to each symbolic label.
read_code(Code, LblMap) :-
   read(Instr),
   read_code(Instr, 0, Code, LblMap).

read_code(end_of_file, _, Code, LblMap) :- !, seal(Code), seal(LblMap).
read_code((:-Option), PC, Code, LblMap) :- !,
   asserta(bamspec_option(Option)),
   read(NextInstr),
   read_code(NextInstr, PC, Code, LblMap).
read_code(Instr, PC, Code, LblMap) :-
   ins(Code, PC, Instr),
   insert_lblmap(Instr, LblMap, PC),
   PC1 is PC+1,
   read(NextInstr),
   read_code(NextInstr, PC1, Code, LblMap).

% Add an entry to the label map:
insert_lblmap(label(L),     LblMap, PC) :- !, ins(LblMap, L, PC).
insert_lblmap(procedure(P), LblMap, PC) :- !, ins(LblMap, P, PC).
insert_lblmap(_, _, _).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Top level execution ***

execute :-
   write('Reading BAM code'), nl,
   read_code(Code, LblMap),
   write('Starting execution'), nl,
   execute(leaf, Regs, leaf, _, leaf, _, Code, _, LblMap, _, 0, N),
   write('Executed '), write(N), write(' instructions.'), nl,
   print_array(Regs).

execute(File) :-
   seeing(OldFile),
   see(File),
   read_code(Code, LblMap),
   seen,
   see(OldFile),
   execute(leaf, Regs, leaf, _, leaf, _, Code, _, LblMap, _, 0, N),
   write('Executed '), write(N), write(' instructions.'), nl,
   print_array(Regs).
```

```
execute -->>
   [set(r(e),leaf)]:regs,
   [set(r(b),leaf)]:regs,
   [set(r(h),tvar^0)]:regs,
   [set(r(tr),0)]:regs,
   [set(r(pc),0)]:regs,
   [set(r(cp),global_success/0)]:regs,
   instr(choice(1/2,[],global_failure/0)),
   instr_loop.

% Instruction execution loop:
instr_loop -->>
   [get(r(pc),PC)]:regs,
   !,
   instr_loop_end(PC).
instr_loop -->>
   error(['Attempt to execute beyond existing code.']).

instr_loop_end(write/1) -->> !, write_rtn, instr(return), instr_loop.
instr_loop_end(nl/0) -->> !, nl, instr(return), instr_loop.
instr_loop_end(global_success/0) -->> !,
   write('*** Global success ***'), nl.
instr_loop_end(global_failure/0) -->> !,
   write('*** Global failure ***'), nl.
instr_loop_end(fail) -->>
   instr(fail),
   !,
   instr_loop.
instr_loop_end(PC) -->>
   numeric_pc(PC, NPC),
   % Fetch:
   [get(NPC,Instr)]:code,
   NPC1 is NPC+1,
   [set(r(pc),NPC1)]:regs,
   % Execute:
   [1]:count,
   flag_print(Instr),
   instr(Instr),
   !,
   instr_loop.
instr_loop_end(PC) -->>
   error(['Program counter is ',PC]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** BAM Instructions ***

% 1. Unification support instructions:
instr(deref(V,W)) -->>
   var_i(get,  V, X),
   deref_rtn(X, Y),
   var_i(set, W, Y).
instr(equal(EA,A,L)) -->>
   ea_e(EA, X),
```

```
   arg_i(A, Y),
   {lbl(L)},
   equal_rtn(X, Y, L).
instr(unify(V,W,F,G,L)) -->>
   var_i(get, V, X),
   var_i(get, W, Y),
   {nv_flag(F)},
   {nv_flag(G)},
   {lbl(L)},
   unify_rtn(X, Y, L).
instr(unify_atomic(V,I,L)) -->>
   var_i(get, V, X),
   {an_atomic(I)},
   {lbl(L)},
   unify_rtn(X, I, L).
instr(trail(V)) -->>
   var_i(get, V, X),
   trail_rtn(X).
instr(move(EA,VI)) -->>
   ea_m(EA, X),
   var_i(set, VI, X), !.
instr(push(EA,R,N)) -->>
   ea_p(EA, X),
   {hreg(R)},
   [get(R,Y)]:regs,
   [set(Y,X)]:heap,
   {pos(N)},
   add_word(Y, N, YN),
   [set(R,YN)]:regs.
instr(adda(R,S,T)) -->>
   {hreg(R)},
   [get(R,X)]:regs,
   numreg(S, Off),
   add_word(X, Off, NX),
   {hreg(T)},
   [set(T,NX)]:regs.
instr(pad(N)) -->>
   [get(r(h),H)]:regs,
   {pos(N)},
   add_word(H, N, NewH),
   [set(r(h),NewH)]:regs.

% 2. Conditional control flow instructions:
instr(choice(1/N,Rs,L)) -->> {pos(N), N>1, regs(Rs), lbl(L)}, !,
   save_choice_regs(Rs, NewB),
   {ins(NewB, r(retry), L)},
   [get(r(tr),TR)]:regs, {ins(NewB, r(tr), TR)},
   [get(r(e),  E)]:regs, {ins(NewB, r(e),   E)},
   [get(r(cp),CP)]:regs, {ins(NewB, r(cp), CP)},
   [get(r(b),  B)]:regs, {ins(NewB, r(b),   B)},
   [get(r(h),  H)]:regs, {ins(NewB, r(h),   H)},
   {seal(NewB)},
   [set(r(hb), H)]:regs,
   [set(r(b),NewB)]:regs.
```

```
instr(choice(I/N,Rs,L)) -->> {pos(N), pos(I), 1<I, I<N, regs(Rs), lbl(L)}, !,
   [get(r(b),B)]:regs,
   restore_choice_regs(Rs, B),
   {set(B,r(retry),L,NewB)},
   [set(r(b),NewB)]:regs.
instr(choice(N/N,Rs,L)) -->> {pos(N), regs(Rs), lbl(L)}, !,
   [get(r(b),B)]:regs,
   restore_choice_regs(Rs, B),
   {get(B,r(b),NewB)},
   [set(r(b),NewB)]:regs,
   {get(NewB,r(h),H)},
   [set(r(hb),H)]:regs.
instr(fail) -->>
   [get(r(b),B)]:regs,
   {get(B,r(h),H)},
   [set(r(h),H)]:regs,
   {get(B,r(e),E)},
   [set(r(e),E)]:regs,
   {get(B,r(cp),CP)},
   [set(r(cp),CP)]:regs,
   [get(r(tr),CurTR)]:regs,
   {get(B,r(tr),OldTR)},
   detrail_rtn(CurTR, OldTR),
   {get(B,r(retry),L)},
   [set(r(pc),L)]:regs.
instr(switch(T,V,A,B,C)) -->>
   {a_tag(T)},
   var_i(get, V, X),
   extract_tag(X, TX),
   {lbl(A), lbl(B), lbl(C)},
   switch_rtn(T, TX, A, B, C).
instr(test(Eq,T,V,L)) -->>
   {a_tag(T)},
   var_i(get, V, X),
   extract_tag(X, TX),
   {eq_ne(Eq)},
   {lbl(L)},
   test_rtn(Eq, T, TX, L).
instr(jump(C,A,B,L)) -->>
   {cond(C)},
   numarg(A, XA), {extract_value(XA, VA), check_int(XA)},
   numarg(B, XB), {extract_value(XB, VB), check_int(XB)},
   {lbl(L)},
   jump_cond_rtn(C, VA, VB, L).
instr(move(r(b),V)) -->>
   [get(r(b),B)]:regs,
   a_var(set, V, B).
instr(cut(V)) -->>
   a_var(get, V, X),
   [set(r(b),X)]:regs,
   {get(X,r(h),H)},
   [set(r(hb),H)]:regs.
instr(hash(T,R,N,L)) -->> hash_type(T), pos(N), lbl(L),
   reg(get, R, X),
```

```
   hash_indirect(T, X, Y),
   [get(L,PC)]:lblmap,
   hash_lookup(PC, Y, N).
instr(pair(_,_)) -->>
   {error(['Attempt to execute inside a hash table.'])}.

% 3. Arithmetic instructions:
instr(add(A,B,V)) -->> arith(add, A, B, V).
instr(sub(A,B,V)) -->> arith(sub, A, B, V).
instr(mul(A,B,V)) -->> arith(mul, A, B, V).
instr(div(A,B,V)) -->> arith(div, A, B, V).
instr(mod(A,B,V)) -->> arith(mod, A, B, V).
instr(and(A,B,V)) -->> arith(and, A, B, V).
instr( or(A,B,V)) -->> arith( or, A, B, V).
instr(xor(A,B,V)) -->> arith(xor, A, B, V).
instr(not(A,V))   -->> arith(not, A, 0, V).
instr(sll(A,B,V)) -->> arith(sll, A, B, V).
instr(sra(A,B,V)) -->> arith(sra, A, B, V).

% 4. Procedural instructions:
instr(procedure(N/A)) -->> {atom(N), natural(A)}.
instr(call(N/A)) -->> {atom(N), natural(A)},
   [get(r(pc),PC)]:regs,
   [set(r(cp),PC)]:regs,
   [set(r(pc),N/A)]:regs.
instr(return) -->>
   [get(r(cp),PC)]:regs,
   [set(r(pc),PC)]:regs.
instr(simple_call(N/A)) -->> {atom(N), natural(A)},
   [get(r(pc),PC)]:regs,
   [set(r(tmp_cp),PC)]:regs,
   [set(r(pc),N/A)]:regs.
instr(simple_return) -->>
   [get(r(tmp_cp),PC)]:regs,
   [set(r(pc),PC)]:regs.
instr(label(L)) -->> {lbl(L)}.
instr(jump(L)) -->> {lbl(L)},
   [set(r(pc),L)]:regs.
instr(allocate(N)) -->>
   {natural(N)},
   [get(r(e),E)]:regs,
   {ins(NewE, r(e), E)},
   [get(r(cp),CP)]:regs,
   {ins(NewE, r(cp), CP)},
   {seal(NewE)},
   [set(r(e),NewE)]:regs.
instr(deallocate(N)) -->>
   {natural(N)},
   [get(r(e),E)]:regs,
   {get(E,r(e),NewE)},
   {get(E,r(cp),NewCP)},
   [set(r(e),NewE)]:regs,
   [set(r(cp),NewCP)]:regs.
instr(nop) -->> [].
```

```
% 5. Pragma information for translator and reorderer:
% Pragmas are no-ops in the execution.
instr(pragma(P)) -->> {pragma(P)}, !.

% 6. Additions to BAM for the assembly language programmer:
% The meaning of these instructions depends on the underlying architecture,
% so they are not included in this specification.  See the Implementation
% Manual for a discussion of their use.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Pragmas ***

% A variable is a multiple of N.
% Inserted just before loads in readmode unification.
pragma(align(V,N))         :- a_var(V), pos(N).

% Inserted just before a sequence of pushes in writemode unification.
% (The pushes may be interleaved with non-memory moves.)
pragma(push(term(Size)))   :- pos(Size).
pragma(push(cons)).
pragma(push(structure(A))) :- pos(A).
pragma(push(variable)).

% Specify the tag of a variable.
% (This is useful for processors without explicit tag support.)
pragma(tag(V,T))           :- a_var(V), a_tag(T).

% Length of a hash table.
pragma(hash_length(Len))   :- pos(Len).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Tags ***

a_tag(tatm). /* atom */
a_tag(tint). /* integer */
a_tag(tneg). /* negative integer */
a_tag(tpos). /* nonnegative integer */
a_tag(tstr). /* structure */
a_tag(tlst). /* cons cell */
a_tag(tvar). /* variable */

atom_tag(tatm).

atomic_tag(tatm).
atomic_tag(tint).
atomic_tag(tneg).
atomic_tag(tpos).

pointer_tag(tstr).
pointer_tag(tlst).
pointer_tag(tvar).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Addressing modes ***

% Both read and write access:

heap(get, W, X) -->> {ptr_word(W)}, [get(W,X)]:heap.
heap(set, W, X) -->> {ptr_word(W)}, [set(W,X)]:heap.

ptr_word(Tˆ_) :- pointer_tag(T).

reg(get, R, X) -->> {reg(R)}, [get(R,X)]:regs.
reg(set, R, X) -->> {reg(R)}, [set(R,X)]:regs.

reg(r(I)) :- int(I), !.

hreg(R) :- reg(R), !.
hreg(r(h)).

perm(get, P, X) -->> {perm(P)}, [get(r(e),E)]:regs, {get(E,P,X)}.
perm(set, P, X) -->> {perm(P)}, [get(r(e),E)]:regs, {set(E,P,X,NewE)},
            [set(r(e),NewE)]:regs.

perm(p(I)) :- natural(I).

a_var(WR, V, X) -->> reg(WR, V, X), !.
a_var(WR, V, X) -->> perm(WR, V, X).

a_var(Reg)  :- reg(Reg), !.
a_var(Perm) :- perm(Perm).

var_i(WR, [V], X) -->> a_var(get, V, W), heap(WR, W, X), !.
var_i(WR,   V, X) -->> a_var(WR, V, X).

% Read access only:

% An int is its own value:
int(N) :- integer(N).

% An atomic is its own value:
an_atomic(I)       :- int(I), !.
an_atomic(TˆA)     :- atom(A), atom_tag(T), !.
an_atomic(Tˆ(F/N)) :- atom(F), pos(N), atom_tag(T).

arg(Arg, Arg) -->> {an_atomic(Arg)}, !.
arg(Arg,   X) -->> a_var(get, Arg, X).

arg_i(Arg, Arg) -->> {an_atomic(Arg)}, !.
arg_i(Arg,   X) -->> var_i(get, Arg, X).

numreg(Arg, Arg) -->> {int(Arg)}, !.
numreg(Arg,   X) -->> reg(get, Arg, X).
```

```
numarg(Arg, Arg) -->> {int(Arg)}, !.
numarg(Arg,   X) -->> var_i(get, Arg, X).

var_off([Var+I], X) -->> a_var(get, Var, T), !,
             {pos(I)}, add_word(T, I, T2), [get(T2,X)]:heap.
var_off([Var],   X) -->> a_var(get, Var, T), [get(T,X)]:heap.

% Creating immediate tagged pointer objects:
imm_tag(Tag^(r(h)+D), W) -->> {pointer_tag(Tag)}, !,
   [get(r(h),T)]:regs,
   {pos(D)}, add_word(T, D, X),
   insert_tag(Tag, X, W).
imm_tag(Tag^r(h), W) -->> {pointer_tag(Tag)}, !,
   [get(r(h),X)]:regs,
   insert_tag(Tag, X, W).

% Effective address for equal:
ea_e(Var,    X) -->> a_var(get, Var, X), !.
ea_e(VarOff, X) -->> var_off(VarOff, X).

% Effective address for move:
ea_m(Arg,    X) -->> arg(Arg, X), !.
ea_m(VarOff, X) -->> var_off(VarOff, X), !.
ea_m(T^r(h), X) -->> imm_tag(T^r(h), X).

% Effective address for push:
ea_p(Arg,    X) -->> arg_i(Arg, X), !.
ea_p(T^Y,    X) -->> imm_tag(T^Y, X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** Miscellaneous ***

eq_ne(eq). /* Equal */
eq_ne(ne). /* Not equal */

cond(lts). /* Signed less than */
cond(les). /* Signed less than or equal */
cond(gts). /* Signed greater than */
cond(ges). /* Signed greater than or equal */
cond(eq).  /* Equal */
cond(ne).  /* Not equal */

hash_type(atomic).
hash_type(structure).

lbl(fail).
lbl(N/A)      :- atom(N), natural(A).
lbl(l(N/A,I)) :- atom(N), natural(A), natural(I).

nv_flag(nonvar).
nv_flag(var).
nv_flag('?').
```

```
% A list of register numbers:
% (May contain the value 'no' as well)
regs([]).
regs([R|Set]) :- (int(R); R=no), !, regs(Set).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Dereference utilities:

deref_rtn(X, X) -->> {nonvartag(X)}, !.
deref_rtn(X, Y) -->>
   [get(X,X2)]:heap,
   deref_rtn_cont(X, X2, Y).

deref_rtn_cont(X, X, Y) -->> !, {Y=X}.
deref_rtn_cont(_, X, Y) -->> deref_rtn(X, Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Equal routine:

equal_rtn(X, X, _) -->> !.
equal_rtn(_, _, L) -->> [set(r(pc),L)]:regs.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Switch and test routines:

switch_rtn(_, tvar, A,_,_) -->> !, [set(r(pc),A)]:regs.
switch_rtn(T,   TX, _,B,_) -->> {equivalent_tag(T,TX)},!, [set(r(pc),B)]:regs.
switch_rtn(_,    _, _,_,C) -->> [set(r(pc),C)]:regs.

test_rtn(Eq, T, TX, L) -->> {test_true(Eq, T, TX)}, !, [set(r(pc),L)]:regs.
test_rtn( _, _,  _, _) -->> [].

test_true(eq, T, TX) :- equivalent_tag(T, TX).
test_true(ne, T, TX) :- \+equivalent_tag(T, TX).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Arithmetic utilities:

arith(Op, A, B, V) -->>
   numarg(A, XA), {extract_value(XA, VA), check_int(XA)},
   numarg(B, XB), {extract_value(XB, VB), check_int(XB)},
   arith_operation(Op, VA, VB, VC),
   a_var(set, V, VC).

arith_operation(add, VA, VB, VC) :- VC is VA+VB.
arith_operation(sub, VA, VB, VC) :- VC is VA-VB.
arith_operation(mul, VA, VB, VC) :- VC is VA*VB.
arith_operation(div, VA, VB, VC) :- VC is VA//VB.
arith_operation(mod, VA, VB, VC) :- VC is VA mod VB.
arith_operation(and, VA, VB, VC) :- VC is VA /\ VB.
```

```
arith_operation( or, VA, VB, VC) :- VC is VA \/ VB.
arith_operation(xor, VA, VB, VC) :- VC is (VA /\ \(VB)) \/ (VB /\ \(VA)).
arith_operation(not, VA,  _, VC) :- VC is \(VA).
arith_operation(sll, VA, VB, VC) :- VC is VA<<VB.
arith_operation(sra, VA, VB, VC) :- VC is VA>>VB.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Conditional jump:

jump_cond_rtn(C, VA, VB, L) -->> {jump_true(C, VA, VB)}, !, [set(r(pc),L)]:regs.
jump_cond_rtn(_,  _,  _, _) -->> [].

jump_true(lts, VA, VB) :- VA@<VB.
jump_true(gts, VA, VB) :- VA@>VB.
jump_true(les, VA, VB) :- VA@=<VB.
jump_true(ges, VA, VB) :- VA@>=VB.
jump_true( eq, VA, VB) :- VA==VB.
jump_true( ne, VA, VB) :- VA\==VB.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Hash table utilities:

hash_lookup(PC, X, N) -->>
   {PC1 is PC+1},
   [get(PC1,pragma(hash_length(N)))]:code,
   {PC2 is PC1+1},
   {PCN is PC1+N},
   hash_lookup_2(PC2, PCN, X).

hash_lookup_2(PC, PCN, _) -->> {PC>PCN}, !.
hash_lookup_2(PC, PCN, X) -->> {PC=<PCN},
   [get(PC,pair(E,L))]:code,
   {E=X},
   !,
   [set(r(pc),L)]:regs.
hash_lookup_2(PC, PCN, X) -->> {PC=<PCN},
   {PC1 is PC+1},
   hash_lookup_2(PC1, PCN, X).

% Indirection needed for structures because main functor is in memory:
hash_indirect(atomic,    X, X) -->> [].
hash_indirect(structure, X, Y) -->> [get(X,Y)]:heap.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Choice point and fail utilities:

save_choice_regs([], _) -->> [].
save_choice_regs([no|Rs], B) -->> !,
   save_choice_regs(Rs, B).
save_choice_regs([I|Rs], B) -->>
   [get(r(I),R)]:regs,
```

```
   {ins(B, r(I), R)},
   save_choice_regs(Rs, B).

restore_choice_regs([], _) -->> [].
restore_choice_regs([no|Rs], B) -->> !,
   restore_choice_regs(Rs, B).
restore_choice_regs([I|Rs], B) -->>
   {get(B, r(I), R)},
   [set(r(I),R)]:regs,
   restore_choice_regs(Rs, B).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Trailing and detrailing:

trail_rtn(X) -->>
   [get(r(hb),HB)]:regs,
   cmp_trail(X, HB).

cmp_trail(X, HB) -->> {less_trail(X, HB)}, !,
   [get(r(tr),TR)]:regs,
   [set(TR,X)]:trail,
   {TR1 is TR+1},
   [set(r(tr),TR1)]:regs.
cmp_trail(_, _) -->> [].

less_trail(_^X, _^Y) :- X<Y.

% Restore to unbound the variables on the trail between OldTR and CurTR.
detrail_rtn(CurTR, OldTR) -->> {CurTR=<OldTR}, !.
detrail_rtn(CurTR, OldTR) -->> {CurTR>OldTR},
   {CurTR1 is CurTR-1},
   [get(CurTR1,V)]:trail,
   [set(V,V)]:heap,
   detrail_rtn(CurTR1, OldTR).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% General unification routine:

unify_rtn(W1, W2, L) -->>
   unify_rtn_2(W1, W2, Flag),
   unify_end(Flag, L).

unify_end(success, _) -->> [].
% For later: detrailing if L\fail.
unify_end(fail, L) -->> [set(r(pc),L)]:regs.

unify_rtn_2(W1, W2, Flag) -->>
   {extract_tag_value(W1, T1, V1)},
   {extract_tag_value(W2, T2, V2)},
   unify_rtn_2(T1, V1, T2, V2, Flag).

unify_rtn_2(tvar, V1, NTag, V2, success) -->> {NTag\==tvar}, !,
```

```
   trail_rtn(tvar^V1),
   {make_word(NTag, V2, Word)},
   [set(tvar^V1,Word)]:heap.
unify_rtn_2(NTag, V2, tvar, V1, success) -->> {NTag\==tvar}, !,
   trail_rtn(tvar^V1),
   {make_word(NTag, V2, Word)},
   [set(tvar^V1,Word)]:heap.
unify_rtn_2(tvar, V1, tvar, V2, success) -->> !,
   unify_varvar(V1, V2).
% Matching atomic tags:
unify_rtn_2(ATag, V1, BTag, V2, Flag) -->>
   {atomic_tag(ATag)},
   {atomic_tag(BTag)},
   {equivalent_tag(ATag, BTag)},
   !,
   unify_atm(V1, V2, Flag).
% Non-matching nonvariable tags:
unify_rtn_2(ATag, _, BTag, _, fail) -->>
   {ATag\==tvar, BTag\==tvar},
   {\+equivalent_tag(ATag, BTag)},
   !.
% Matching pointer tags (recursive case):
unify_rtn_2(ATag, V1, ATag, V2, Flag) -->>
   {pointer_tag(ATag)},
   get_size(ATag, V1, Sz),
   unify_rtn_args_2(0, Sz, ATag, V1, V2, Flag).

% The term’s Size is the maximum offset needed to traverse the term in memory.
get_size(tlst, _, 1) -->> [].
get_size(tstr, V, N) -->>
   [get(tstr^V,Func)]:heap,
   {Func=(tatm^(_/N))}.

unify_rtn_args_2(N, Sz, _, _, _, success) -->> {N>Sz}, !.
unify_rtn_args_2(N, Sz, T, V, W, Flag) -->> {N=<Sz}, !,
   {VN is V+N},
   {WN is W+N},
   [get(T^VN,VX)]:heap, deref_rtn(VX, DVX),
   [get(T^WN,WX)]:heap, deref_rtn(WX, DWX),
   unify_rtn_2(DVX, DWX, F),
   {N1 is N+1},
   unify_rtn_args_3(F, N1, Sz, T, V, W, Flag).

% Continue with other arguments if argument unification succeeded:
unify_rtn_args_3(fail, _, _, _, _, _, fail) -->> [].
unify_rtn_args_3(success, N1, Sz, T, V, W, Flag) -->>
   unify_rtn_args_2(N1, Sz, T, V, W, Flag).

% Unifying value parts of two atomic terms with equivalent tag:
unify_atm(V, V, success) -->> !.
unify_atm(_, _, fail) -->> [].

% Unifying two variables: bind youngest to oldest, trail youngest.
unify_varvar(V1, V2) -->> {V1>V2}, !,
```

```
   trail_rtn(tvar^V1),
   [set(tvar^V1,tvar^V2)]:heap.
unify_varvar(V1, V2) -->> {V1=<V2}, !,
   trail_rtn(tvar^V2),
   [set(tvar^V2,tvar^V1)]:heap.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Simple type utilities:

ground(X) :- nonvar(X), functor(X, _, N), ground(N, X).

ground(N, _) :- N=:=0, !.
ground(N, X) :- N=\=0, arg(N, X, A), ground(A), N1 is N-1, ground(N1, X).

natural(N) :- integer(N), N>=0.
pos(N)  :- integer(N), N>0.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Word, tag, and value manipulation utilities:

% This takes into account the relationship between tpos, tneg and tint.
% For integers it extracts tpos or tneg tags and the absolute value
% of the integer.  It creates the correct integer, given the tpos, tneg
% or tint tags.

equivalent_tag(T,       T) :- !.
equivalent_tag(tint, tpos) :- !.
equivalent_tag(tint, tneg).

extract_tag(N, tpos) :- integer(N), N>=0, !.
extract_tag(N, tneg) :- integer(N), N<0, !.
extract_tag(T^_,  T).

extract_value(N,   N) :- int(N), N>=0, !.
extract_value(N,   M) :- int(N), N<0, !, M is -N.
extract_value(_^V, V).

extract_tag_value(W, T, V) :-
   extract_tag(W, T),
   extract_value(W, V).

nonvartag(I) :- int(I), !.
nonvartag(T^_) :- \+T=tvar.

% Only used for pointer tags:
insert_tag(T, _^V, T^V).

make_word(tint, I, I) :- !.
make_word(tpos, I, I) :- !.
make_word(tneg, N, I) :- !, I is -N.
make_word(T, V, T^V).
```

```
add_word(TˆI, J, TˆK) :- K is I+J.

% Eventually, print out value of PC:
check_int(I) :- int(I), !.
check_int(_) :-
   error([’Operand of conditional is not an integer.’]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Table utilities:

% This code implements a mutable array, represented as a binary tree.

% Insert a value in logarithmic time and constant space:
% This predicate is used in this program only to create the array,
% although it can also be used to access array elements.
ins(T, I, V) :- hash(I, H), ins_2(T, H, V).

ins_2(node(N,W,L,R), I, V) :- ins_2(N, W, L, R, I, V).

ins_2(N, V, _, _, I, V) :- I=N, !.
ins_2(N, _, L, R, I, V) :-
        compare(Order, I, N),
        ins_2(Order, I, V, L, R).

ins_2(<, I, V, L, _) :- ins_2(L, I, V).
ins_2(>, I, V, _, R) :- ins_2(R, I, V).

% Access a value in logarithmic time and constant space:
% This predicate cannot be used to create the array incrementally,
% but it is faster than ins/3.
get(T, I, V) :- hash(I, H), get_2(T, H, V).

get_2(node(N,W,L,R), I, V) :-
        compare(Order, I, N),
        get_3(Order, I, V, W, L, R).

get_3(<, I, V, _, L, _) :- get_2(L, I, V).
get_3(=, _, V, W, _, _) :- V=W.
get_3(>, I, V, _, _, R) :- get_2(R, I, V).

% Update an array in logarithmic time and space:
set(T, I, V, U) :- hash(I, H), set_2(T, H, V, U).

set_2(leaf,          I, V, node(I,V,leaf,leaf)).
set_2(node(N,W,L,R), I, V, node(N,NW,NL,NR)) :-
        compare(Order, I, N),
        set_3(Order, I, V, W, L, R, NW, NL, NR).

set_3(<, I, V, W, L, R, W, NL, R) :- set_2(L, I, V, NL).
set_3(=, _, V, _, L, R, V, L, R).
set_3(>, I, V, W, L, R, W, L, NR) :- set_2(R, I, V, NR).

% Prevent any further insertions in the array:
```

```
seal(leaf).
seal(node(_,_,L,R)) :- seal(L), seal(R).

% Print values of array in sorted order:
print_array(Term) :-
   flat_array(Term, 2, Flat),
   print_list(Flat).

print_list([]).
print_list([(A->B)|L]) :-
   write(A), put(9), write('= '), write(B), nl,
   print_list(L).

flat_array(Term, N, Sort) :-
   N>0, N1 is N-1,
   flat_array(Term, N1, Flat, []), !,
   sort(Flat, Sort).
flat_array(leaf,          N,    []) :- N=:=0, !.
flat_array(node(_,_,_,_), N, '...') :- N=:=0, !.
flat_array(Term, _, Term).

flat_array(leaf, _) --> [].
flat_array(node(H,T,L,R), N) -->
   flat_array(L, N),
   {hash(H, I)},
   {flat_array(T, N, F)},
   [(I->F)],
   flat_array(R, N).

% Invertible hash function:
% Bit inversion of the integer components of a ground term.  Other parts are
% unchanged.  This one inverts the low 16 bits.  It can be changed by changing
% the last argument of bit_invert/3.
hash(I, H) :- integer(I), !, bit_invert(I, H, 16).
hash(T, H) :- functor(T, Na, Ar), functor(H, Na, Ar), hash_2(Ar, T, H).

hash_2(0, _, _) :- !.
hash_2(N, T, H) :- N>0,
   arg(N, T, X),
   arg(N, H, Y),
   hash(X, Y),
   N1 is N-1,
   hash_2(N1, T, H).

bit_invert(0, 0, _) :- !.
bit_invert(N, I, B) :- N>0,
   L is N>>1,
   R is N/\1,
   B1 is B-1,
   bit_invert(L, LI, B1),
   I is R*(1<<B) + LI.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Error handling:

error(L) :-
   write('*** Error: '),
   error_loop(L),
   write(' ***'), nl.

error_loop([]).
error_loop([M|L]) :- write(M), error_loop(L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Primitive version of write:

write_rtn -->>
   [get(r(0),X)]:regs,
   write_rtn(X).

write_rtn(tvar^V) -->> !, {write('_'), write(V)}.
write_rtn(I) -->> {int(I)}, !, {write(I)}.
write_rtn(tatm^(F/N)) -->> !, {write(''''), write(F/N), write('''')}.
write_rtn(tatm^A) -->> !, {write(A)}.
write_rtn(tlst^V) -->> !,
   {W is V+1},
   [get(tlst^V,Head)]:heap,
   [get(tlst^W,Tail)]:heap,
   deref_rtn(Head, DHead),
   deref_rtn(Tail, DTail),
   {write('[')},
   write_rtn(DHead),
   {write('|')},
   write_rtn(DTail),
   {write(']')}.
write_rtn(tstr^V) -->> !,
   [get(tstr^V,tatm^(F/N))]:heap,
   {write(F), write('(')},
   write_arg(V, 1),
   write_args(2, N, V),
   {write(')')}.

write_args(I, N, _) -->> {I>N}, !.
write_args(I, N, V) -->> {I=<N}, !,
   {I1 is I+1},
   {write(',')},
   write_arg(V, I),
   write_args(I1, N, V).

write_arg(V, I) -->>
   {W is V+I},
   [get(tstr^W,X)]:heap,
   deref_rtn(X, DX),
   write_rtn(DX).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```