Warren's Abstract Machine

A Tutorial Reconstruction

Hassan Aït-Kaci

ICLP'91 Pre-Conference Tutorial

I dedicate this modest work to a most outstanding

W A M

I am referring - of course - to

Wolfgang Amadeus Mozart

1756 - 1791

Year's to you...

-hak

Introduction

Warren's Abstract Machine (WAM) was specified in 1983 by David H. D. Warren [6].

Until recently, there was no clear account of its workings.

This course is entirely based on the instructor's recent monograph [1]:

- It consists of a *gradual* reconstruction of the WAM through several intermediate abstract machine designs.
- It is a *complete* account justifying *all* design features.

Course Outline:

- Unification
- Flat resolution
- Pure Prolog
- Optimizations

Unification—Pure and Simple

First-order term

- a *variable*, denoted by a capitalized identifier;
  (*e.g.*, $X, X1, Y, Constant, \ldots$);
- a *constant* denoted by an identifier starting with a lower-case letter;
  (*e.g.*, $a, b, variable, cONSTANT, \ldots$);
- a *structure* of the form $f(t_1, \ldots, t_n)$ where f is a symbol called a *functor* (denoted like a constant), and the t_i's are first-order terms;
  (*e.g.*, $f(X), f(a, g(X, h(Y), Y), g(X)), \ldots$).

'f/n' denotes the functor with symbol f and arity n.

A constant c is a special case of a structure with functor $c/0$.

Language $\mathcal{L}_0$

- **Syntax:**
  two syntactic entities:
  - a *program* term, noted t;
  - a *query* term, noted $?{-}t$;

  where t is a *non-variable* first-order term. (the scope of variables is limited to a program (resp., a query) term.)

- **Semantics:**
  computation of the MGU of the program p and the query $?{-}q$; having specified p, submit $?{-}q$,
  - either execution fails if p and q do not unify;
  - or it succeeds with a binding of the variables in q obtained by unifying it with p.

In $\mathcal{L}_0$, failure aborts all further work.

Abstract machine $\mathcal{M}_0$

Heap representation of terms:

$\mathcal{M}_0$ uses a global storage area called `HEAP`, an array of data cells, to represent terms internally:

0	`STR`	1
1	$h/2$	
2	`REF`	2
3	`REF`	3
4	`STR`	5
5	$f/1$	
6	`REF`	3
7	`STR`	8
8	$p/3$	
9	`REF`	2
10	`STR`	1
11	`STR`	5

Representation of $p(Z, h(Z, W), f(W))$.
starting at heap address 7.

Heap data cells:

- *variable cell:*

  $\langle$ `REF` , k $\rangle$, where k is a store address; *i.e.*, an index into `HEAP`;

- *structure cell:*

  $\langle$ `STR` , k $\rangle$, where k where is the address of a functor cell;

- *functor cell:*

  (untagged) contains the representation of a functor.

Convention:

- An *unbound variable* at address k is $\langle$ `REF` , k $\rangle$.
- A structure $f(t_1, \ldots, t_n)$ takes $n + 2$ heap cells.
- The first cell of $f(t_1, \ldots, t_n)$ is $\langle$ `STR` , k $\rangle$, where k is the address of a (possibly non-contiguous) functor cell containing f/n.
- A functor cell is *always* immediately followed by of n contiguous cells; *i.e.*, if `HEAP[`k`]` = f/n then `HEAP[`$k+1$`]` refers to (t_1), ..., and `HEAP[`$k+n$`]` to (t_n).

Compiling $\mathcal{L}_0$ queries

Preparing one side of an equation to be solved.

Namely, a query term $?{-}q$ is translated into a sequence of instructions designed to build an exemplar of q on the heap from q's textual form.

Variable registers

$\mathrm{X}1, \mathrm{X}2, \ldots,$ are used to store temporarily heap data cells as terms are being built.

They are allocated to a term, one for each subterms.

Convention:

- Variable registers are allocated according to least available index.
- Register $\mathrm{X}1$ is always allocated to the outermost term.
- A same register is allocated to all the occurrences of a given variable.

Registers allocated to the term $p(Z, h(Z, W), f(W))$:

$$\begin{aligned}
\mathrm{X}1 &= p(\mathrm{X}2, \mathrm{X}3, \mathrm{X}4) \\
\mathrm{X}2 &= Z \\
\mathrm{X}3 &= h(\mathrm{X}2, \mathrm{X}5) \\
\mathrm{X}4 &= f(\mathrm{X}5) \\
\mathrm{X}5 &= W.
\end{aligned}$$

Flattened form

A term is equivalent to a conjunctive set of equations of the form $X_i = X$ or $X_i = f(X_{i_1}, \ldots, X_{i_n})$, $(n \geq 0)$ where the X_i's are all distinct new variable names.

- external variable names are meaningless;
- a query term's *flattened form* is a sequence of register assignments of the form

$$\mathrm{X}i = f(\mathrm{X}i_1, \ldots, \mathrm{X}i_n)$$

  ordered from the bottom up; *i.e.*, so that a register is assigned *before* it is used as an argument as a subterm.

The flattened form of query term $p(Z, h(Z, W), f(W))$ is:

$$\mathrm{X}3 = h(\mathrm{X}2, \mathrm{X}5), \mathrm{X}4 = f(\mathrm{X}5), \mathrm{X}1 = p(\mathrm{X}2, \mathrm{X}3, \mathrm{X}4).$$

Tokenized form

Scanning a flattened query term from left to right, each $\mathrm{X}i = f(\mathrm{X}i_1, \ldots, \mathrm{X}i_n)$ is tokenized as a sequence $\mathrm{X}i = f/n$, $\mathrm{X}i_1, \ldots, \mathrm{X}i_n$.

The *tokenized form* of query term $p(Z, h(Z, W), f(W))$ is a stream of 9 tokens:

$$\mathrm{X}3 = h/3, \mathrm{X}2, \mathrm{X}5, \mathrm{X}4 = f/1, \mathrm{X}5, \mathrm{X}1 = p/3, \mathrm{X}2, \mathrm{X}3, \mathrm{X}4.$$

There are three kinds of tokens to process:

1. a register associated with a structure functor;
2. a register argument not previously encountered anywhere in the stream;
3. a register argument seen before in the stream.

$\mathcal{M}_0$ query term instructions

Respectively, each of the three token kinds indicates a different action:

1. `put_structure` f/n, `X`i
   push a new `STR` (and adjoining functor) cell onto the heap and copy that cell into the allocated register address;

2. `set_variable` `X`i
   push a new `REF` cell onto the heap containing its own address, and copy it into the given register;

3. `set_value` `X`i
   push a new cell onto the heap and copy into it the register's value.

Heap Register: **H**

H keeps the address of the next free cell in the heap.

$$
\begin{aligned}
\texttt{put\_structure}\ f/n, \texttt{X}i \equiv\ & \texttt{HEAP}[\mathbf{H}] \leftarrow \langle\, \texttt{STR}\,, \mathbf{H}+1 \,\rangle; \\
& \texttt{HEAP}[\mathbf{H}+1] \leftarrow f/n; \\
& \texttt{X}i \leftarrow \texttt{HEAP}[\mathbf{H}]; \\
& \mathbf{H} \leftarrow \mathbf{H}+2;
\end{aligned}
$$

$$
\begin{aligned}
\texttt{set\_variable}\ \texttt{X}i \equiv\ & \texttt{HEAP}[\mathbf{H}] \leftarrow \langle\, \texttt{REF}\,, \mathbf{H} \,\rangle; \\
& \texttt{X}i \leftarrow \texttt{HEAP}[\mathbf{H}]; \\
& \mathbf{H} \leftarrow \mathbf{H}+1;
\end{aligned}
$$

$$
\begin{aligned}
\texttt{set\_value}\ \texttt{X}i \equiv\ & \texttt{HEAP}[\mathbf{H}] \leftarrow \texttt{X}i; \\
& \mathbf{H} \leftarrow \mathbf{H}+1;
\end{aligned}
$$

$\mathcal{M}_0$ machine instructions for query terms

```
put_structure h/2, X3   %  ?-X3 = h
set_variable X2         %         (Z,
set_variable X5         %           W),
put_structure f/1, X4   %     X4 = f
set_value X5            %         (W),
put_structure p/3, X1   %     X1 = p
set_value X2            %         (Z,
set_value X3            %           X3,
set_value X4            %             X4).
```

$\mathcal{M}_0$ machine code for $\mathcal{L}_0$ query

?$-p(Z, h(Z, W), f(W))$.

Compiling $\mathcal{L}_0$ programs

Compiling a program term p assumes that a query `?-`q has been built a term on the heap and set register `X1` to contain its address.

Therefore, code for an $\mathcal{L}_0$ program term uses two *modes:*

- a `READ` mode in which data on the heap is matched against;
- a `WRITE` mode in which a term is built on the heap exactly as is a query term.

Code for p consists of:

- following the term structure already present in `X1` as long as it matches functor for functor the structure of p;
- when an unbound `REF` cell is encountered in the query term `?-`q in the heap, then it is bound to a new term that is built on the heap as an exemplar of the corresponding subterm in p.

Tokenizing $\mathcal{L}_0$ program term

Variable registers are allocated as before; *e.g.*, for program term $p(f(X), h(Y, f(a)), Y)$:

$$\begin{aligned}
\mathrm{X}1 &= p(\mathrm{X}2, \mathrm{X}3, \mathrm{X}4)\\
\mathrm{X}2 &= f(\mathrm{X}5)\\
\mathrm{X}3 &= h(\mathrm{X}4, \mathrm{X}6)\\
\mathrm{X}4 &= Y\\
\mathrm{X}5 &= X\\
\mathrm{X}6 &= f(\mathrm{X}7)\\
\mathrm{X}7 &= a.
\end{aligned}$$

But now the the flattened form follows a *, top down* order because query data from the heap are assumed available (even if only in the form of unbound `REF` cells).

Program term $p(f(X), h(Y, f(a)), Y)$ is flattened into:

$$\mathrm{X}1 = p(\mathrm{X}2, \mathrm{X}3, \mathrm{X}4), \mathrm{X}2 = f(\mathrm{X}5),$$

$$\mathrm{X}3 = h(\mathrm{X}4, \mathrm{X}6), \mathrm{X}6 = f(\mathrm{X}7), \mathrm{X}7 = a.$$

Tokenizing this is just as before.

$\mathcal{M}_0$ query term instructions

Program tokens correspond to three kinds of machine instructions:

1. `get_structure` f/n, Xi
2. `unify_variable` Xi
3. `unify_value` Xi

depending on whether is met, respectively:

1. a register associated with a structure functor;
2. a first-seen register argument;
3. an already-seen register argument.

```
get_structure p/3, X1   %  X1 = p
unify_variable X2       %       (X2,
unify_variable X3       %           X3,
unify_variable X4       %               Y),
get_structure f/1, X2   %  X2 = f
unify_variable X5       %       (X),
get_structure h/2, X3   %  X3 = h
unify_value X4          %       (Y,
unify_variable X6       %           X6),
get_structure f/1, X6   %  X6 = f
unify_variable X7       %       (X7),
get_structure a/0, X7   %  X7 = a.
```

$\mathcal{M}_0$ machine code for $\mathcal{L}_0$ program

$$p(f(X), h(Y, f(a)), Y).$$

Dereferencing

Variable binding creates reference chains.

Dereferencing is performed by a function $deref$ which, when applied to a store address, follows a possible reference chain until it reaches either an unbound `REF` cell or a non-`REF` cell, the address of which it returns.

`READ`/`WRITE` mode

The two `unify` instructions work in two modes depending on whether a term is to be matched from, or being built on, the heap.

- For building (`WRITE` mode), they work exactly like the two `set` query instructions.
- For matching (`READ` mode), they seek to recognize data from the heap as those of the term at corresponding positions, proceeding if successful and failing otherwise.

Subterm Register: **S**

S keeps the heap address of the next subterm to be matched in `READ` mode.

Mode is set by `get_structure` f/n, `X`i:

- if $deref($`X`$i)$ is a `REF` cell (*i.e.*, unbound variable), then binds to a new `STR` cell pointing to f/n pushed onto the heap and mode is set to `WRITE`;
- otherwise,
  - if it is an `STR` cell pointing to functor f/n, then register **s** is set to the heap address following that functor cell's and mode is set to `READ`.
  - If it is not an `STR` cell or if the functor is not f/n, the program fails.

```
get_structure f/n, Xi
        ≡ addr ← deref(Xi);
          case STORE[addr] of
            ⟨ REF , _ ⟩ : HEAP[H] ← ⟨ STR , H + 1 ⟩;
                          HEAP[H + 1] ← f/n;
                          bind(addr, H);
                          H ← H + 2;
                          mode ← WRITE;
            ⟨ STR , a ⟩ : if HEAP[a] = f/n
                           then
                            begin
                             S ← a + 1;
                             mode ← READ
                            end
                           else fail ← true;
             other      : fail ← true;
          endcase;
```

$\mathcal{M}_0$ machine instruction `get_structure`

`unify_variable` Xi:

- in `READ` mode, sets register Xi to the contents of the heap at address **s**;
- in `WRITE` mode, a new unbound `REF` cell is pushed on the heap and copied into Xi.

In both modes, **s** is then incremented by one.

`unify_value` Xi:

- in `READ` mode, the value of Xi must be unified with the heap term at address **s**;
- in `WRITE` mode, a new cell is pushed onto the heap and set to the value of register Xi.

Again, in either mode, **s** is incremented.

$$
\begin{aligned}
\texttt{unify\_variable}\ \texttt{X}i \equiv\ & \textbf{case}\ \textit{mode}\ \textbf{of} \\
& \quad \texttt{READ}\ \ : \texttt{X}i \leftarrow \texttt{HEAP}[\mathbf{S}]; \\
& \quad \texttt{WRITE}: \texttt{HEAP}[\mathbf{H}] \leftarrow \langle \texttt{REF}, \mathbf{H} \rangle; \\
& \qquad\qquad\ \ \texttt{X}i \leftarrow \texttt{HEAP}[\mathbf{H}]; \\
& \qquad\qquad\ \ \mathbf{H} \leftarrow \mathbf{H} + 1; \\
& \textbf{endcase}; \\
& \mathbf{S} \leftarrow \mathbf{S} + 1;
\end{aligned}
$$

$$
\begin{aligned}
\texttt{unify\_value}\ \texttt{X}i \equiv\ & \textbf{case}\ \textit{mode}\ \textbf{of} \\
& \quad \texttt{READ}\ \ : \textit{unify}(\texttt{X}i, \mathbf{S}); \\
& \quad \texttt{WRITE}: \texttt{HEAP}[\mathbf{H}] \leftarrow \texttt{X}i; \\
& \qquad\qquad\ \ \mathbf{H} \leftarrow \mathbf{H} + 1; \\
& \textbf{endcase}; \\
& \mathbf{S} \leftarrow \mathbf{S} + 1;
\end{aligned}
$$

$\mathcal{M}_0$ `unify` machine instructions

Variable Binding

bind is performed on two store addresses, at least one of which is that of an unbound `REF` cell.

For now:

- it binds the unbound one to the other—*i.e.*, change the data field of the unbound `REF` cell to contain the address of the other cell;
- if both arguments are unbound `REF`'s, the binding direction is chosen arbitrarily.

NOTE: *bind* may also perform an *occurs-check* test in order to prevent formation of cyclic terms — by failing at that point.

```
procedure unify(a1, a2 : address);
  push(a1, PDL); push(a2, PDL);
  fail ← false;
  while ¬(empty(PDL) ∨ fail) do
    begin
      d1 ← deref(pop(PDL)); d2 ← deref(pop(PDL));
      if d1 ≠ d2 then
        begin
          ⟨ t1 , v1 ⟩ ← STORE[d1]; ⟨ t2 , v2 ⟩ ← STORE[d2];
          if (t1 = REF) ∨ (t2 = REF)
            then bind(d1, d2)
            else
              begin
                f1/n1 ← STORE[v1]; f2/n2 ← STORE[v2];
                if (f1 = f2) ∧ (n1 = n2)
                  then
                    for i ← 1 to n1 do
                      begin
                        push(v1 + i, PDL); push(v2 + i, PDL)
                      end
                  else fail ← true
              end
        end
    end
end unify;
```

Language $\mathcal{L}_1$

We now make a distinction between:

- *atoms* (terms whose functor is a predicate); and,
- *terms* (arguments to a predicate).

Extend $\mathcal{L}_0$ into $\mathcal{L}_1$:

- **Syntax:**
  similar to $\mathcal{L}_0$ but now a program may be a *set of first-order atoms* each defining at most one *fact* per predicate name.

- **Semantics:**
  execution of a query connects to the appropriate definition to use for solving a given unification equation, or fails if none exists for the predicate invoked.

The set of instructions $\mathcal{I}_1$ contains all those in $\mathcal{I}_0$.

In $\mathcal{M}_1$, compiled code is stored in a *code area* (`CODE`), an array of possibly labeled instructions consisting of an opcode followed by operands.

The size of an instruction stored at address a (*i.e.*, `CODE[`a`]`) is given by the expression $instruction\_size(a)$.

Labels are symbolic entry points into the code area that may be used as operands of instructions for transferring control to the code labeled accordingly.

Therefore, *there is no need to store a procedure name in the heap as it denotes a key into a compiled instruction sequence.*

Control Instructions

The standard execution order of instructions is sequential.

Program Register: `P`

> `P` keeps the address of the next instruction to execute.

Unless failure occurs, most machine instructions are implicitly assumed, to increment `P` by $instruction\_size(\texttt{P})$. Some instructions break sequential execution or connect to some other instruction at the end of a sequence.

These instructions are called *control instructions* as they typically set `P` in a non-standard way.

$\mathcal{M}_1$'s control instructions are:

- `call` $p/n \equiv \mathbf{P} \leftarrow @(p/n)$;
  where $@(p/n)$ is the address in the code area of instruction labeled p/n. If the procedure p/n is not defined, failure occurs and overall execution aborts.

- `proceed`
  indicates the end of a fact's instruction sequence.

Argument registers

In $\mathcal{L}_1$, unification between fact and query terms amounts to solving, not one, but many equations, simultaneously.

As $\mathtt{X1}$ in $\mathcal{M}_0$ always contains the (single) term root, in $\mathcal{M}_1$ registers $\mathtt{X1}, \ldots, \mathtt{X}n$ are systematically allocated to contain the roots of the n arguments of an n-ary predicate.

Then, we speak of *argument registers,* and we write $\mathtt{A}i$ rather than $\mathtt{X}i$ when the i-th register contains the i-th argument.

Where register $\mathtt{X}i$ is not used as an argument register, it is written $\mathtt{X}i$, as usual. (NOTE: this is just notation—the $\mathtt{A}i$'s and the $\mathtt{X}i$'s are the same.)

e.g., for atom $p(Z, h(Z, W), f(W))$, $\mathcal{M}_1$ allocates registers:

$$\begin{aligned}\mathtt{A1} &= Z\\ \mathtt{A2} &= h(\mathtt{A1}, \mathtt{X4})\\ \mathtt{A3} &= f(\mathtt{X4})\\ \mathtt{X4} &= W.\end{aligned}$$

Argument instructions

These are needed in $\mathcal{M}_1$ to handle variable arguments.

As in $\mathcal{L}_0$, instructions correspond to when a variable argument is a first or later occurrence, either in a query or a fact.

In a query:

- the first occurrence of a variable in i-th argument position pushes a new unbound `REF` cell onto the heap and copies it into that variable's register as well as argument register `A`i;
- a later occurrence copies its value into argument register `A`i.

In a fact:

- the first occurrence of a variable in i-th argument position sets it to the value of argument register `A`i;
- a later occurrence unifies it with the value of `A`i.

The corresponding instructions, respectively:

$$
\begin{array}{lll}
\texttt{put\_variable}\ \texttt{X}n, \texttt{A}i & \equiv & \texttt{HEAP}\,[\mathbf{H}] \leftarrow \langle\, \texttt{REF}\ , \mathbf{H} \,\rangle; \\
 & & \texttt{X}n \leftarrow \texttt{HEAP}\,[\mathbf{H}]; \\
 & & \texttt{A}i \leftarrow \texttt{HEAP}\,[\mathbf{H}]; \\
 & & \mathbf{H} \leftarrow \mathbf{H} + 1; \\
 & & \\
\texttt{put\_value}\ \texttt{X}n, \texttt{A}i & \equiv & \texttt{A}i \leftarrow \texttt{X}n \\
 & & \\
\texttt{get\_variable}\ \texttt{X}n, \texttt{A}i & \equiv & \texttt{X}n \leftarrow \texttt{A}i \\
 & & \\
\texttt{get\_value}\ \texttt{X}n, \texttt{A}i & \equiv & \mathit{unify}(\texttt{X}n, \texttt{A}i)
\end{array}
$$

$\mathcal{M}_1$ instructions for variable arguments

```
put_variable X4, A1        %  ?-p(Z,
put_structure h/2, A2      %        h
set_value X4               %          (Z,
set_variable X5            %             W),
put_structure f/1, A3      %                f
set_value X5               %                  (W)
call p/3                   %                    ).
```

Argument registers for $\mathcal{L}_1$ query

$?\text{-}p(Z, h(Z, W), f(W)).$

```
p/3 : get_structure f/1, A1   %  p(f
      unify_variable X4       %      (X),
      get_structure h/2, A2   %          h
      unify_variable X5       %            (Y,
      unify_variable X6       %               X6),
      get_value X5, A3        %                  Y),
      get_structure f/1, X6   %  X6 = f
      unify_variable X7       %          (X7),
      get_structure a/0, X7   %  X7 = a
      proceed                 %  .
```

Argument registers for $\mathcal{L}_1$ fact $p(f(X), h(Y, f(a)), Y)$.

Language $\mathcal{L}_2$: Flat Resolution

$\mathcal{L}_2$ is Prolog without backtracking:

- it extends $\mathcal{L}_1$ with procedures which are no longer reduced only to facts but may also have bodies;
- a body defines a procedure as a conjunctive sequence of atoms;
- there is at most one defining clause per predicate name.

Syntax of $\mathcal{L}_2$

An $\mathcal{L}_2$ *program* is a set of procedure definitions of the form 'a_0 :- $a_1, \ldots, a_n$.' where $n \geq 0$ and the a_i's are atoms.

As before, when $n = 0$, the clause is called a *fact* and written without the '**:-**' implication symbol.

When $n > 0$, the clause is called a *rule*.

A rule with exactly one body goal is called a *chain* (rule).

Other rules are called *deep* rules.

An $\mathcal{L}_2$ query is a sequence of goals, of the form '?-$g_1, \ldots, g_k$.' where $k \geq 0$.

As in Prolog, the scope of variables is limited to the clause or query in which they appear.

Semantics of $\mathcal{L}_2$

Executing a query '$?- g_1, \ldots, g_k.$' in the context of a program made up of a set of procedure-defining clauses consists of repeated application of *leftmost resolution* until the empty query, or failure, is obtained.

Leftmost resolution:

- unify the goal g_1 with its definition's head (or failing if none exists); then,
- if this succeeds, transform the query replacing g_1 by its definition body, variables in scope bearing the binding side-effects of unification.

Therefore, executing a query in $\mathcal{L}_2$ either:

- terminates with success; or,
- terminates with failure; or,
- never terminates.

The "result" of an $\mathcal{L}_2$ query whose execution terminates with success is the (dereferenced) binding of its original variables after termination.

Compiling $\mathcal{L}_2$

To compile an $\mathcal{L}_2$ clause head, $\mathcal{M}_1$'s fact instructions are sufficient.

As a first approximation, compiled code for a query (resp., a clause body) is the concatenation of the compiled code of each goal as an $\mathcal{L}_1$ query.

However, $\mathcal{M}_2$ must take two measures of caution regarding:

- continuation of execution of a goal sequence;
- avoiding conflicts in the use of argument registers.

$\mathcal{L}_2$ Facts

Now `proceed` must continue execution, after successfully returning from a call to a fact, back to the instruction in the goal sequence following the call.

Continuation Point Register: CP

> **CP** is used by $\mathcal{M}_2$ to save and restore the address of the next instruction to follow up with upon successful return from a call.

Thus, for $\mathcal{L}_2$'s facts, $\mathcal{M}_2$ alters $\mathcal{M}_1$'s control instructions to:

$$\texttt{call}\ p/n \equiv \begin{array}{l} \mathbf{CP} \leftarrow \mathbf{P} + \mathit{instruction\_size}(\mathbf{P}); \\ \mathbf{P} \leftarrow @(p/n); \end{array}$$

$$\texttt{proceed} \equiv \mathbf{P} \leftarrow \mathbf{CP};$$

As before, when the procedure p/n is not defined, execution fails.

With this simple adjustment, $\mathcal{L}_2$ facts are translated exactly as were $\mathcal{L}_1$ facts.

Rules and queries

As first approximation, translate a rule

$$p_0(\ldots) \texttt{ :- } p_1(\ldots), \ldots, p_n(\ldots).$$

following the pattern:

get arguments of p_0
put arguments of p_1
`call` p_1
⋮
put arguments of p_n
`call` p_n

(The case of a query is the particular case of a rule with no head instructions.)

- Variables which occur in more than one body goal are called *permanent* as they have to outlive the procedure call where they first appear.
- All other variables in a scope that are not permanent are called *temporary*.

Problem:

Because the same variable registers are used by every body goal, permanent variables run the risk of being overwritten by intervening goals.

e.g., in

$$p(X, Y) \text{ :- } q(X, Z), r(Z, Y).$$

no guarantee can be made that the variables Y, Z are still in registers after executing q.

NOTE: To determine whether a variable is permanent or temporary in a rule, the head atom is considered to be part of the first body goal (*e.g.*, X in example above is temporary).

Solution:

Save temporary variables in an *environment* associated with each activation of the procedure they appear in.

$\mathcal{M}_2$ saves a procedure's permanent variables and register **CP** in a run-time stack, a data area (called STACK), of procedure activation frames called *environments*.

Environment Register: **E**

E keeps the address of the latest environment on STACK.

$\mathcal{M}_2$'s STACK is organized as a linked list of frames of the form:

E	CE (*previous environment*)
E + 1	**CP** (*continuation point*)
E + 2	n (*number of permanent variables*)
E + 3	Y1 (*permanent variable 1*)
	⋮
E + n + 2	Yn (*permanent variable* n)

(We write a permanent variable as Yi, and use Xi as before for temporary variables.)

An environment is pushed onto `STACK` upon a (non-fact) procedure entry call, and popped from `STACK` upon return; *i.e.*, $\mathcal{L}_2$ rule:

$$p_0(\ldots) \text{ :- } p_1(\ldots), \ldots, p_n(\ldots).$$

is translated in $\mathcal{M}_2$ code:

`allocate` N
get arguments of p_0
put arguments of p_1
`call` p_1
⋮
put arguments of p_n
`call` p_n
`deallocate`

- `allocate` N
  create and pushe an environment frame for N permanent variables onto `STACK`;

- `deallocate`
  discard the environment frame on top of `STACK` and set execution to continue at continuation point recovered from the environment being discarded.

That is,

$$
\begin{aligned}
\texttt{allocate}\ N \equiv\ & newE \leftarrow \mathbf{E} + \texttt{STACK}[\mathbf{E}+2] + 3; \\
& \texttt{STACK}[newE] \leftarrow \mathbf{E}; \\
& \texttt{STACK}[newE+1] \leftarrow \mathbf{CP}; \\
& \texttt{STACK}[newE+2] \leftarrow N; \\
& \mathbf{E} \leftarrow newE; \\
& \mathbf{P} \leftarrow \mathbf{P} + instruction\_size(\mathbf{P});
\end{aligned}
$$

$$\equiv$$

$$
\begin{aligned}
\texttt{deallocate} \equiv\ & \mathbf{P} \leftarrow \texttt{STACK}[\mathbf{E}+1]; \\
& \mathbf{E} \leftarrow \texttt{STACK}[\mathbf{E}];
\end{aligned}
$$

```
p/2 : allocate 2              %  p
      get_variable X3, A1     %    (X,
      get_variable Y1, A2     %       Y) :-
      put_value X3, A1        %            q(X,
      put_variable Y2, A2     %              Z
      call q/2                %                ),
      put_value Y2, A1        %            r(Z,
      put_value Y1, A2        %              Y
      call r/2                %                )
      deallocate              %                .
```

$\mathcal{M}_2$ machine code for rule $p(X,Y)$:- $q(X,Z), r(Z,Y)$.

Language $\mathcal{L}_3$: Pure Prolog

Syntax of $\mathcal{L}_3$

- $\mathcal{L}_3$ extends the language $\mathcal{L}_2$ to allow disjunctive definitions.
- As in $\mathcal{L}_2$, an $\mathcal{L}_3$ program is a set of procedure definitions.
- In $\mathcal{L}_3$, a definition is an ordered sequence of clauses (*i.e.*, a sequence of facts or rules) consisting of all and only those whose head atoms share the same predicate name — the name of the procedure specified by the definition.
- $\mathcal{L}_3$ queries are the same as those of $\mathcal{L}_2$.

Semantics of $\mathcal{L}_3$

- operates using top-down leftmost resolution, an approximation of SLD resolution.
- failure of unification no longer yields irrevocable abortion of execution but considers alternative choices by chronological backtracking; *i.e.*, the latest choice at the moment of failure is reexamined first.

$\mathcal{M}_3$ alters $\mathcal{M}_2$'s design so as to save the state of computation at each procedure call offering alternatives.

We call such a state a *choice point*:

> It contains all relevant information needed for a correct state of computation to be restored to try the next alternative, with all effects of the failed computation undone.

$\mathcal{M}_3$ manages choice points as frames in a stack (just like environments).

To distinguish the two stacks, we call the environment stack the *AND-stack* and the choice point stack the *OR-stack*.

Backtrack Register: B

`B` keeps the address of the latest choice point.

- upon failure, computation is resumed from the state recovered from the choice point frame indicated by `B`;
- if the frame offers no more alternatives, it is popped off the OR-stack by resetting `B` to its predecessor if one exists; otherwise, computation fails terminally.

NOTE: if a definition contains only one clause, there is no need to create a choice point frame, exactly as was the case in $\mathcal{M}_2$.

For definitions with more than one alternative,

- a choice point frame is created by the first alternative;
- then, it is updated (as far as which alternative to try next) by intermediate (but non ultimate) alternatives;
- finally, it is discarded by the last alternative.

Environment protection

Problem:

In (deterministic) $\mathcal{L}_2$, it is safe for $\mathcal{M}_2$ to deallocate an environment frame at the end of a rule.

This is no longer true for $\mathcal{M}_3$: later failure may force reconsidering a choice from a computation state in the middle of a rule whose environment has long been deallocated.

Example

Program:

$$a :- b(X), c(X).$$

$$b(X) :- e(X).$$

$$c(1).$$

$$e(X) :- f(X).$$

$$e(X) :- g(X).$$

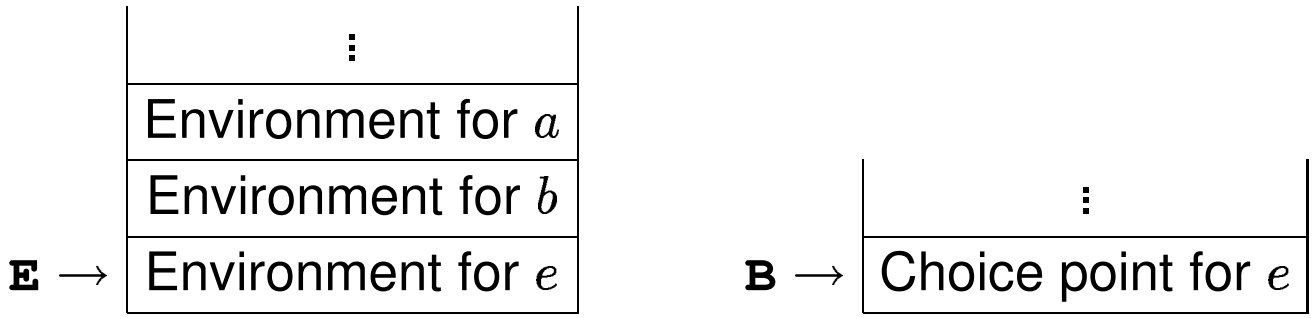
$$f(2).$$

$$g(1).$$

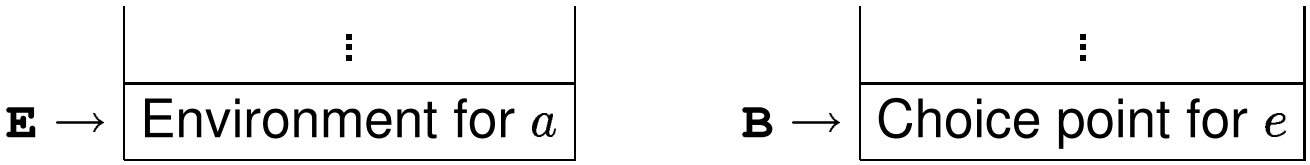
Query:

$$?- a.$$

- allocate environment for a;
- call b;
- allocate environment for b;
- call e:
  - create and push choice point for e;
  - allocate environment for e;



- call f;
- succeed $(X = 2)$;
- deallocate environment for e;
- deallocate environment for b;



Continuing with execution of a's body:

- call c;
- failure $(X = 2 \neq 1)$;

The choice point indicated by **`B`** shows an alternative clause for e, but at this point b*'s environment has been lost.*

$\mathcal{M}_3$ must prevent unrecoverable deallocation of environment frames that chronologically precede any existing choice point.

IDEA: every choice point must "protect" from deallocation all environment frames existing before its creation.

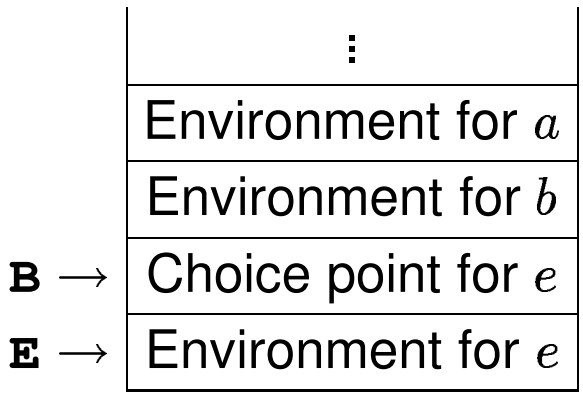
Solution:

$\mathcal{M}_3$ uses the same stack for *both* environments and choice points: a choice point now caps all older environments:

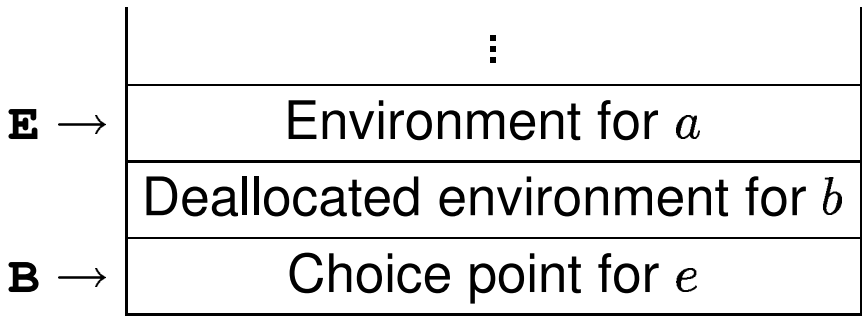
- As long as a choice point is active, it forces allocation of further environments on top of it, precluding overwriting of the (even explicitly deallocated) older environments.
- Safe resurrection of a deallocated protected environment is automatic when coming back to an alternative from this choice point.
- Protection lasts just as long as it is needed: as soon as the choice point disappears, all explicitly deallocated environments may be safely overwritten.

Back to our example:

- allocate environment for a;
- call b;
- allocate environment for b;
- call e:
  - create and push choice point for e;
  - allocate environment for e;



- call f;
- succeed $(X = 2)$;
- deallocate environment for e;
- deallocate environment for b;



Continuing with execution of a's body:

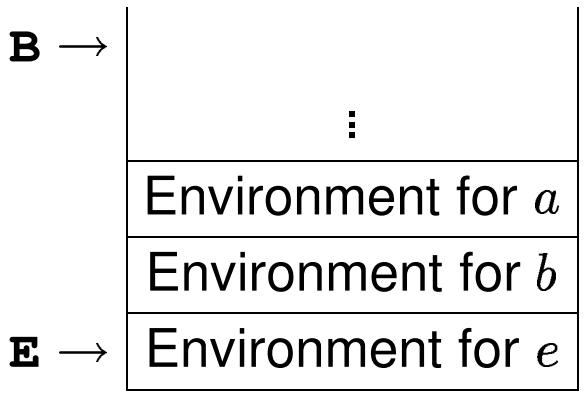
- call c;
- failure $(X = 2 \neq 1)$;

Now, $\mathcal{M}_3$ can safely recover the state from the choice point for e indicated by **B**, in which the saved environment to restore is the one current at the time of this choice point's creation—*i.e.*, that (still existing) of b.

- backtrack;
- discard choice point for e;

Protection is now (safely) ended.

Execution of the last alternative for e proceeds with:



Undoing bindings

Binding effects must be undone when reconsidering a choice.

$\mathcal{M}_3$ records in a data area called the *trail* (`TRAIL`) all variables which need to be reset to 'unbound' upon backtracking.

Trail Register: **`TR`**

> **`TR`** keeps the next available address on `TRAIL`.

NOTE: Only *conditional* bindings need to be trailed.

A conditional binding is one affecting a variable existing before creation of the current choice point.

Heap Backtrack Register: **`HB`**

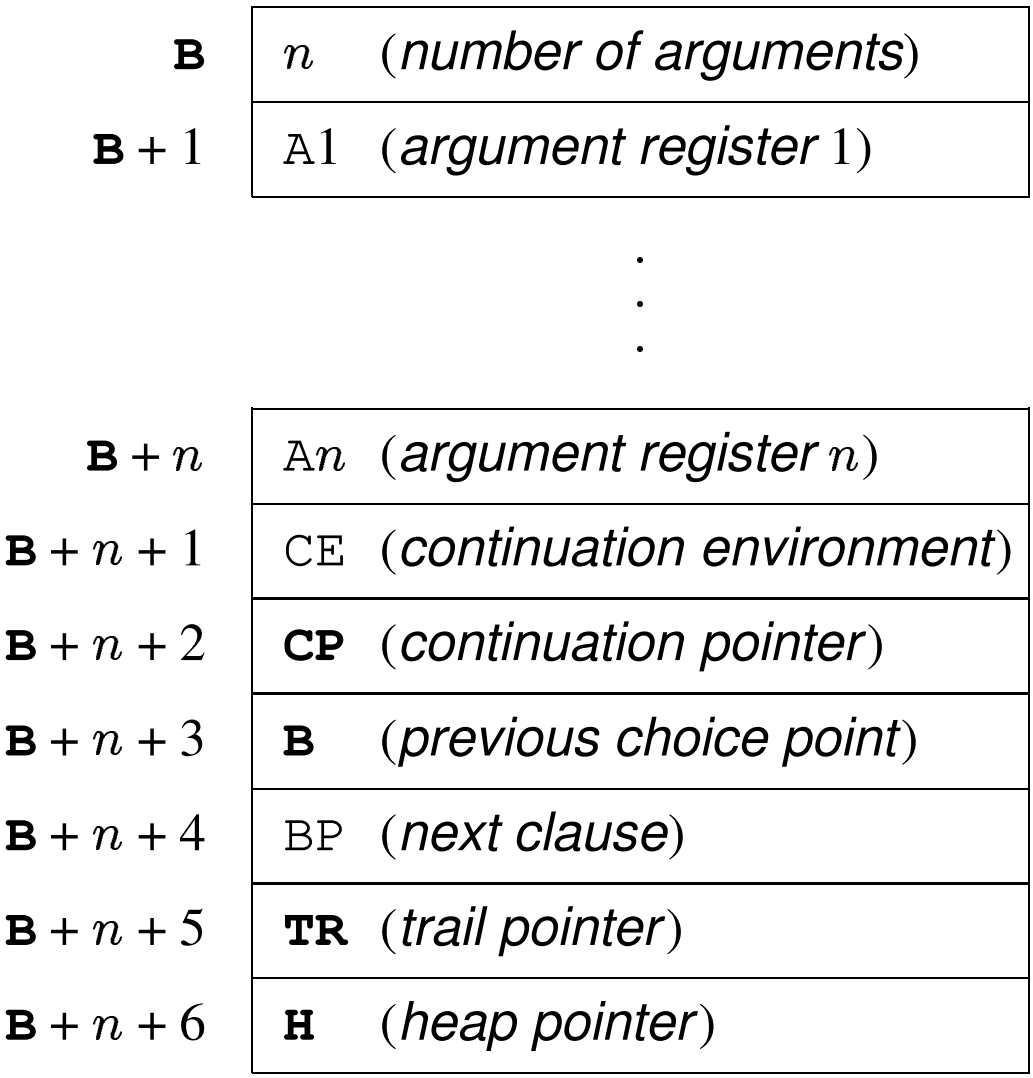
> **`HB`** keeps the value of **`H`** at the time of the latest choice point's creation.

- `HEAP[`a`]` is conditional *iff* $a <$ **`HB`**;
- `STACK[`a`]` is conditional *iff* $a <$ **`B`**.

What's in a choice point?

- *The argument registers* `A1`*, ...,* `A`n, where n is the arity of the procedure offering alternative choices of definitions.
- *The current environment* (value of register **`E`**), to recover as a protected environment.
- *The continuation pointer* (value of register **`CP`**), as the current choice will overwrite it.
- *The latest choice point* (value of register **`B`**), where to backtrack in case all alternatives offered by the current choice point fail.
- *The next clause*, to try in this definition in case the currently chosen one fails. This slot is updated at each backtracking to this choice point if more alternatives exist.
- *The current trail pointer* (value of register **`TR`**), which is needed as the boundary where to *unwind* the trail upon backtracking.
- *The current top of heap* (value of register **`H`**), which is needed to recover (garbage) heap space of all the structures and variables constructed during the failed attempt.

Choice point frame:



NOTE: $\mathcal{M}_3$ must alter $\mathcal{M}_2$'s definition of `allocate` to:

$$
\begin{aligned}
\texttt{allocate}\ N \equiv\ & \textbf{if } \texttt{E} > \texttt{B} \\
& \quad \textbf{then } newE \leftarrow \texttt{E} + \texttt{STACK}[\texttt{E} + 2] + 3 \\
& \quad \textbf{else } newE \leftarrow \texttt{B} + \texttt{STACK}[\texttt{B}] + 7; \\
& \texttt{STACK}[newE] \leftarrow \texttt{E}; \\
& \texttt{STACK}[newE + 1] \leftarrow \texttt{CP}; \\
& \texttt{STACK}[newE + 2] \leftarrow N; \\
& \texttt{E} \leftarrow newE; \\
& \texttt{P} \leftarrow \texttt{P} + instruction\_size(\texttt{P});
\end{aligned}
$$

Choice instructions

Given a multiple-clause definition, $\mathcal{M}_3$ use three instructions to deal with, respectively:

1. the first clause;
2. an intermediate (but non ultimate) clause;
3. the last clause.

They are, respectively:

1. `try_me_else` L
   allocate a new choice point frame on the stack setting its next clause field to L and the other fields according to the current context, and set **B** to point to it;
2. `retry_me_else` L
   reset all the necessary information from the current choice point and update its next clause field to L;
3. `trust_me`
   reset all the necessary information from the current choice point, then discard it by resetting **B** to the value of its predecessor slot.

Bactracking

In $\mathcal{M}_3$, all $\mathcal{M}_2$ instructions where failure may occur (*i.e.*, some unification instructions and all procedure calls) are altered to end with a test checking whether failure has indeed occurred and, if such is the case, to perform the following operation:

$$backtrack \equiv \texttt{P} \leftarrow \texttt{STACK[B} + \texttt{STACK[B]} + 4\texttt{];}$$

as opposed to setting **P** unconditionally to follow the normal sequence.

If there is no more choice point on the stack, this is a terminal failure and execution aborts.

Recapitulation of $\mathcal{L}_3$ compilation

- The $\mathcal{M}_3$ code generated for a single-clause definition in $\mathcal{L}_3$ is identical to what is generated for an $\mathcal{L}_2$ program on $\mathcal{M}_2$.

- For a two-clause definition for a procedure p/n, the pattern is:

p/n : `try_me_else` L
code for first clause
L : `trust_me`
code for second clause

- and for more than two clauses:

$$
\begin{array}{lll}
p/n & : & \texttt{try\_me\_else}\ L_1 \\
 & & \textit{code for first clause} \\
L_1 & : & \texttt{retry\_me\_else}\ L_2 \\
 & & \textit{code for second clause} \\
 & & \vdots \\
L_{k-1} & : & \texttt{retry\_me\_else}\ L_k \\
 & & \textit{code for penultimate clause} \\
L_k & : & \texttt{trust\_me} \\
 & & \textit{code for last clause}
\end{array}
$$

  where each clause is translated as it would be as a single $\mathcal{L}_2$ clause for $\mathcal{M}_2$.

Example,

$$
\begin{array}{l}
p(X, a). \\
p(b, X). \\
p(X, Y) \texttt{ :- } p(X, a), p(b, Y).
\end{array}
$$

```
p/2 : try_me_else L1        %  p
      get_variable X3, A1   %    (X,
      get_structure a/0, A2 %       a)
      proceed               %         .

L1  : retry_me_else L2      %  p
      get_structure b/0, A1 %    (b,
      get_variable X3, A2   %       X)
      proceed               %         .

L2  : trust_me              %
      allocate 1            %  p
      get_variable X3, A1   %    (X,
      get_variable Y1, A2   %       Y) :-
      put_value X3, A1      %             p(X,
      put_structure a/0, A2 %               a
      call p/2              %                 ),
      put_structure b/0, A1 %             p(b,
      put_value Y1, A2      %               Y
      call p/2              %                 )
      deallocate            %                  .
```

$\mathcal{M}_3$ code for a multiple-clause procedure

Optimizing the Design

WAM Principle 1 *Heap space is to be used as sparingly as possible, as terms built on the heap turn out to be relatively persistent.*

WAM Principle 2 *Registers must be allocated in such a way as to avoid unnecessary data movement, and minimize code size as well.*

WAM Principle 3 *Particular situations that occur very often, even though correctly handled by general-case instructions, are to be accommodated by special ones if space and/or time may be saved thanks to their specificity.*

Heap representation

A better heap representation for $p(Z, h(Z, W), f(W))$ is:

0	$h/2$	
1	REF	1
2	REF	2
3	$f/1$	
4	REF	2
5	$p/3$	
6	REF	1
7	STR	0
8	STR	3

provided that all reference to it from the store or registers is a cell of the form $\langle$ STR , 5 $\rangle$.

Hence, there is actually no need to allot a systematic STR cell before each functor cell.

For this, need only change put_structure to:

$$\begin{aligned} \texttt{put\_structure}\ f/n, \texttt{X}i \equiv\ & \texttt{HEAP}[\mathbf{H}] \leftarrow f/n; \\ & \texttt{X}i \leftarrow \langle\, \texttt{STR}\,, \mathbf{H} \,\rangle; \\ & \mathbf{H} \leftarrow \mathbf{H} + 1; \end{aligned}$$

Constants, lists, and anonymous variables

Constants

```
unify_variable Xi
get_structure c/0, Xi
```

is simplified into one specialized instruction:

```
unify_constant c
```

and

```
put_structure c/0, Xi
set_variable Xi
```

is simplified into:

```
set_constant c
```

Similarly, `put` and `get` instructions can also be simplified from those of structures to deal specifically with constants.

We need a new sort of data cell tagged `CON`, indicating a constant.

e.g., heap representation starting at address 10 for the structure $f(b, g(a))$:

8	$g/1$	
9	CON	a
10	$f/2$	
11	CON	b
12	STR	8

Heap space for a constant is saved when loading a register with it, or binding a variable to it: it is treated as a literal value.

Constant-handling instructions:

- `put_constant` c, Xi
- `get_constant` c, Xi
- `set_constant` c
- `unify_constant` c

```
put_constant c, Xi ≡ Xi ← ⟨ CON , c ⟩;

get_constant c, Xi ≡
        addr ← deref(Xi);
        case STORE[addr] of
          ⟨ REF , _ ⟩  : STORE[addr] ← ⟨ CON , c ⟩;
                         trail(addr);
          ⟨ CON , c' ⟩ : fail ← (c ≠ c');
          other        : fail ← true;
        endcase;

set_constant c    ≡ HEAP[H] ← ⟨ CON , c ⟩;
                    H ← H + 1;

unify_constant c ≡
        case mode of
          READ  : addr ← deref(S);
                  case STORE[addr] of
                    ⟨ REF , _ ⟩  : STORE[addr] ← ⟨ CON , c ⟩;
                                   trail(addr);
                    ⟨ CON , c' ⟩ : fail ← (c ≠ c');
                    other        : fail ← true;
                  endcase;
          WRITE : HEAP[H] ← ⟨ CON , c ⟩;
                  H ← H + 1;
        endcase;
```

Lists

Non-empty list functors need not be represented explicitly on the heap.

Use tag `LIS` to indicate that a cell containss the heap address of the first of a list pair.

List-handling instructions:

```
put_list Xi ≡ Xi ← ⟨ LIS , H ⟩;

get_list Xi ≡ addr ← deref(Xi);
              case STORE[addr] of
                ⟨ REF , _ ⟩ : HEAP[H] ← ⟨ LIS , H + 1 ⟩;
                              bind(addr, H);
                              H ← H + 1;
                              mode ← WRITE;
                ⟨ LIS , a ⟩ : S ← a;
                              mode ← READ;
                other       : fail ← true;
              endcase;
```

```
put_list X5                  %  ?-X5 = [
set_variable X6              %         W|
set_constant []              %           []],
put_variable X4, A1          %    p(Z,
put_list A2                  %      [
set_value X4                 %       Z|
set_value X5                 %         X5],
put_structure f/1, A3        %           f
set_value X6                 %            (W)
call p/3                     %              ).
```

Specialized code for query $?\text{-}p(Z, [Z, W], f(W))$.

```
p/3 : get_structure f/1, A1   %  p(f
      unify_variable X4       %     (X,
      get_list A2             %       [
      unify_variable X5       %        Y|
      unify_variable X6       %          X6],
      get_value X5, A3        %             Y),
      get_list X6             %  X6 = [
      unify_variable X7       %       X7|
      unify_constant []       %         []],
      get_structure f/1, X7   %  X7 = f
      unify_constant a        %       (a)
      proceed                 %         .
```

Specialized code for fact $p(f(X), [Y, f(a)], Y).$

Anonymous variables

A single-occurrence variable in a non-argument positions needs no register.

If many occur in a row as in $f(\_, \_, \_)$ they can be all be processed in one swoop.

Anonymous variable instructions:

- `set_void` n
  push n new unbound `REF` cells on the heap;

- `unify_void` n
  in `WRITE` mode, behave like `set_void` n;
  in `READ` mode, skip the next n heap cells starting at location **s**.

$$
\begin{aligned}
\texttt{set\_void}\ n \quad &\equiv \textbf{for } i \leftarrow \textbf{H} \textbf{ to } \textbf{H} + n - 1 \textbf{ do} \\
&\quad\ \texttt{HEAP}[i] \leftarrow \langle \texttt{REF}, i \rangle; \\
&\quad\ \textbf{H} \leftarrow \textbf{H} + n;
\end{aligned}
$$

$$
\begin{aligned}
\texttt{unify\_void}\ n \equiv\ & \textbf{case } \textit{mode} \textbf{ of} \\
& \quad \texttt{READ}\ \ : \textbf{S} \leftarrow \textbf{S} + n; \\
& \quad \texttt{WRITE}: \textbf{for } i \leftarrow \textbf{H} \textbf{ to } \textbf{H} + n - 1 \textbf{ do} \\
& \qquad\qquad\ \texttt{HEAP}[i] \leftarrow \langle \texttt{REF}, i \rangle; \\
& \qquad\qquad \textbf{H} \leftarrow \textbf{H} + n; \\
& \textbf{endcase}
\end{aligned}
$$

NOTE: an anonymous head argument is simply ignored; since,

$$\texttt{get\_variable}\ \texttt{X}i, \texttt{A}i$$

is clearly vacuous.

```
p/3 : get_structure g/1,A2   %  p(_,g
      unify_void 1           %       (X),
      get_structure f/3,A3   %          f
      unify_void 3           %            (_,Y,_)
      proceed                %                  ).
```

Instructions for fact $p(\_, g(X), f(\_, Y, \_))$.

Register allocation

Clever register allocation allows peep-hole optimization.

e.g., code for fact $conc([], L, L)$. is:

```
conc/3 : get_constant [], A1    % conc([],
         get_variable X4, A2    %          L,
         get_value X4, A3       %            L)
         proceed                %              .
```

It is silly to use X4 for variable L: use A2!

⇒ `get_variable A2, A2` is a no-op and can be eliminated:

```
conc/3 : get_constant [], A1    % conc([],
         get_value A2, A3       %          L, L)
         proceed                %             .
```

Generally, allocate registers so vacuous operations:

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

may be eliminated.

(See [2] for more.)

```
p/2 : allocate 2              %  p
      get_variable X3, A1     %    (X,
      get_variable Y1, A2     %       Y) :-
      put_value X3, A1        %            q(X,
      put_variable Y2, A2     %              Z
      call q/2                %                ),
      put_value Y2, A1        %            r(Z,
      put_value Y1, A2        %              Y
      call r/2                %                )
      deallocate              %                .
```

Naïve code for $p(X, Y) :- q(X, Z), r(Z, Y).$

```
p/2 : allocate 2          %  p
      get_variable Y1, A2 %    (X, Y) :-
      put_variable Y2, A2 %                q(X, Z
      call q/2            %                     ),
      put_value Y2, A1    %                r(Z,
      put_value Y1, A2    %                   Y
      call r/2            %                     )
      deallocate          %                      .
```

Better register use for $p(X, Y) :- q(X, Z), r(Z, Y).$

Last call optimization

LCO generalizes tail-recursion optimization as a stack frame recovery process.

IDEA: Permanent variables are no longer needed after all the `put` instructions preceding the last `call` in the body.

⇒ Discard the current environment *before* the last `call` in a rule's body.

SIMPLE: Just swap the `call`, `deallocate` sequence that always conclude a rule's instruction sequence (*i.e.*, into `deallocate`, `call`).

CAUTION: `deallocate` is no longer the last instruction; so it must reset **CP**, rather than **P**:

$$\begin{aligned}\texttt{deallocate} \equiv\ & \mathbf{CP} \leftarrow \texttt{STACK[}\mathbf{E}+1\texttt{]};\\ & \mathbf{E} \leftarrow \texttt{STACK[}\mathbf{E}\texttt{]};\\ & \mathbf{P} \leftarrow \mathbf{P} + \mathit{instruction\_size}(\mathbf{P})\end{aligned}$$

CAUTION: But when `call` *is* the last instruction, it must not set **CP** but **P**.

So we cannot modify `call`, since it is correct when not last.

For last call, use `execute` p/n:

$$\texttt{execute}\ p/n \equiv\ \mathbf{P} \leftarrow @(p/n);$$

```
p/2 : allocate 2              %  p
      get_variable Y1, A2     %    (X, Y) :-
      put_variable Y2, A2     %                q(X, Z
      call q/2                %                     ),
      put_value Y2, A1        %                r(Z,
      put_value Y1, A2        %                   Y
      deallocate              %                     )
      execute r/2             %                      .
```

$p(X, Y)$:- $q(X, Z), r(Z, Y)$. **with LCO**

Chain rules

Applying LCO, translating a chain rule of the form

$$p(\ldots) \texttt{ :- } q(\ldots).$$

gives:

p : `allocate` N
get arguments of p
put arguments of q
`deallocate`
`execute` q

But all variables in a chain rule are necessarily temporary.

⇒ With LCO, `allocate`/`deallocate` are *useless* in a chain rule — Eliminate them!

i.e., translate a chain rule of the form

$$p(\ldots) \texttt{ :- } q(\ldots).$$

as:

p : *get arguments of* p
put arguments of q
`execute` q

Chain rules need no stack frame at all!

Environment trimming

Sharpens LCO: discard a permanent variable as soon as it is no longer needed.

⇒ The current environment frame will shrink gradually, until it eventually vanishes altogether by LCO.

Rank the PV's of a rule: the later a PV's last goal, the lower its offset in the current environment frame.

e.g., in

$$p(X, Y, Z) \text{ :- } q(U, V, W), r(Y, Z, U), s(U, W), t(X, V).$$

all variables are permanent:

Variable	*Last goal*	*Offset*
X	t	`Y1`
Y	r	`Y5`
Z	r	`Y6`
U	s	`Y3`
V	t	`Y2`
W	s	`Y4`

Now `call` takes a second argument counting the number of PV's still needed after the call.

CAUTION: Modify `allocate` to reflect always a correct stack offset.

FACT: *the CP field of the environment,* `STACK[`**E** + 1`]`*, always contains the address of the instruction immediately following the* `call` P, N *where* N *is the desired offset.*

⇒ `allocate` no longer needs its argument *and* environments no longer need an offset field.

E	`CE` (*continuation environment*)
E + 1	**CP** (*continuation point*)
E + 2	`Y1` (*permanent variable* 1)
	⋮

Alter `allocate` to retrieve the correct trimmed offset as `CODE[STACK[`$\mathbf{E}+1$`]` $-\ 1$`]`:

```
allocate ≡
        if E > B
          then newE ← E + CODE[STACK[E + 1] − 1] + 2
          else newE ← B + STACK[B] + 7;
        STACK[newE] ← E;
        STACK[newE + 1] ← CP;
        E ← newE;
        P ← P + instruction_size(P);
```

(Similarly for `try_me_else`...)

```
p/3 : allocate                % p
      get_variable Y1,A1      %   (X,
      get_variable Y5,A2      %      Y,
      get_variable Y6,A3      %         Z) :-
      put_variable Y3,A1      %                q(U,
      put_variable Y2,A2      %                   V,
      put_variable Y4,A3      %                      W
      call q/3,6              %                        ),
      put_value Y5,A1         %                r(Y,
      put_value Y6,A2         %                   Z,
      put_value Y3,A3         %                      U
      call r/3,4              %                        ),
      put_value Y3,A1         %                s(U,
      put_value Y4,A2         %                   W
      call s/2,2              %                     ),
      put_value Y1,A1         %                t(X,
      put_value Y2,A2         %                   V
      deallocate              %                     )
      execute t/2             %                      .
```

Environment trimming code

Stack variables

A PV $\texttt{Y}n$ that first occurs in the body of a rule as a goal argument is initialized with a $\texttt{put\_variable Y}n, \texttt{A}i$.

This *systematically* sets both $\texttt{Y}n$ and argument register $\texttt{A}i$ to point to a new cell on `HEAP`.

⇒ Modify `put_variable` to work differently on PV's so not to allocate a heap cell as for TV's.

i.e.,

$$
\begin{aligned}
&\texttt{put\_variable Y}n, \texttt{A}i \equiv \\
&\qquad addr \leftarrow \mathbf{E} + n + 1; \\
&\qquad \texttt{STACK}[addr] \leftarrow \langle\, \texttt{REF}\,, addr \,\rangle; \\
&\qquad \texttt{A}i \leftarrow \texttt{STACK}[addr];
\end{aligned}
$$

Unfortunately, there are rather insidious consequences to this apparently innocuous change as it interferes with ET and LCO.

Trouble

PV's may be discarded (by LCO and ET) while still unbound.

⇒ DANGER: risk of dangling references!

e.g.,

- it is incorrect for *bind* to choose an arbitrary pointer direction between two unbound variables.
- some instructions are now incorrect if used blindly in some situations: `put_value` and `set_value` (thus also `unify_value` in `WRITE` mode).

Treatment

- keep a correct binding convention;
- analyze what is wrong with `put_value`, `set_value`, and `unify_value` to avert trouble on the fly – *i.e.*, only when *really* needed.

Variable binding and memory layout

As it turns out, most correct bindings can be ensured following a simple chronological reference rule:

WAM Binding Rule 1 *Always make the variable of higher address reference that of lower address.*

In other words, an older (less recently created) variable cannot reference a younger (more recently created) variable.

Benefit of WAM Binding Rule 1

Three possibilities of variable-variable bindings:

(1) heap-heap,

(2) stack-stack,

(3) heap-stack.

- **Case (1):** unconditional bindings are favored over conditional ones:

  ⇒ no unnecessary trailing;

  ⇒ swift heap space recovery upon backtracking.

- **Case (2):** same applies, but also works consistently with PV ranking for ET within an environment.

  Unfortunately, this is not sufficient to prevent all danger of dangling references.

- **Case (3):** references to `STACK` are unsafe; also need:

  WAM Binding Rule 2 *Heap variables must never be set to a reference into the stack;*

  and follow a specific memory layout convention make this naturally consistent with WAM Binding Rule 1:

  WAM Binding Rule 3 *The stack must be allocated at higher addresses than the heap, in the same global address space.*

Unsafe variables

Remaining problem

WAM Binding Rule 2 can *still* be violated by `put_value`, `set_value`, and `unify_value`.

A PV which is initialized by a `put_variable` (*i.e.*, which first occurs as the argument of a body goal) is called *unsafe*.

e.g., in

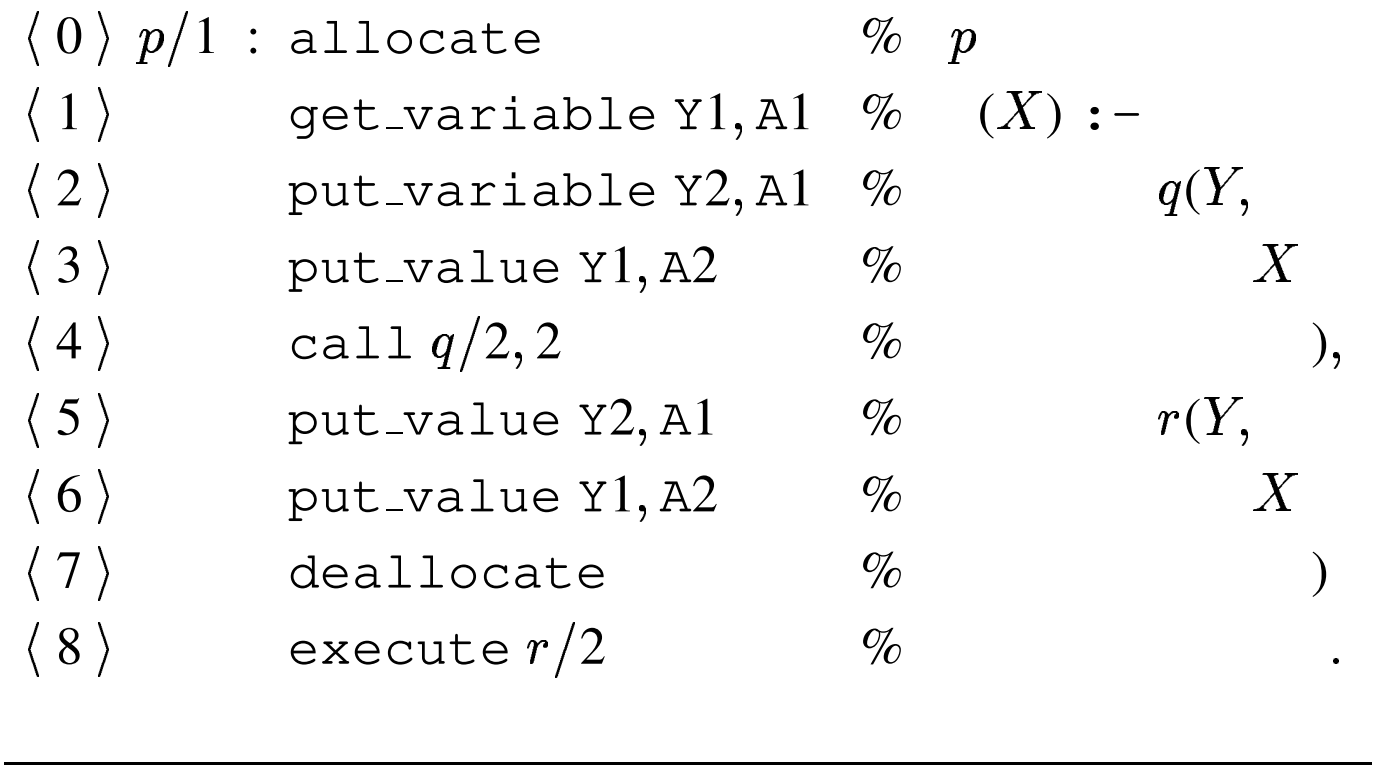
$$p(X) :- q(Y, X), r(Y, X).$$

both X and Y are PV's, but only Y is unsafe.

Assume p is called with an unbound argument;

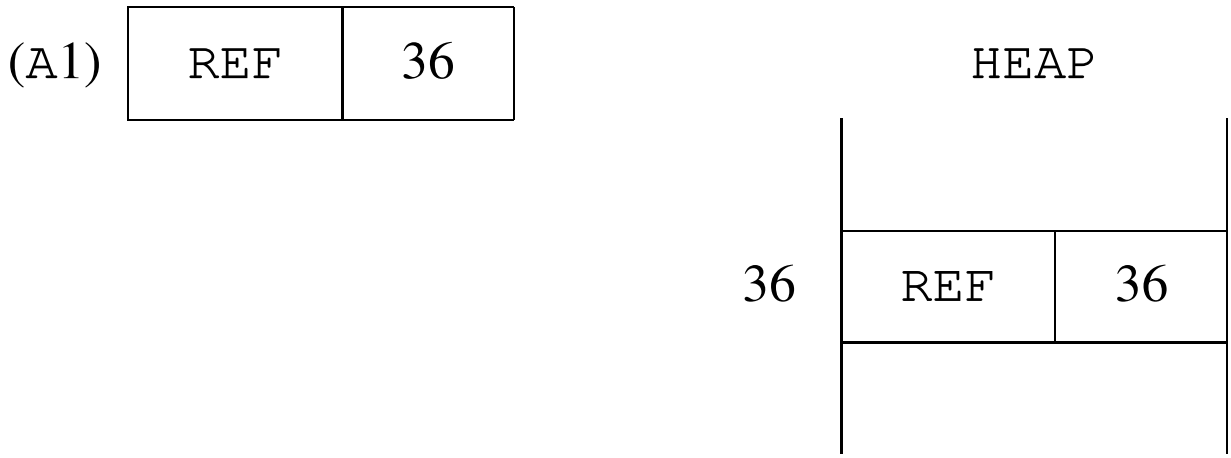
e.g.,

`put_variable` $\texttt{X}i, \texttt{A1}$
`execute` $p/1$

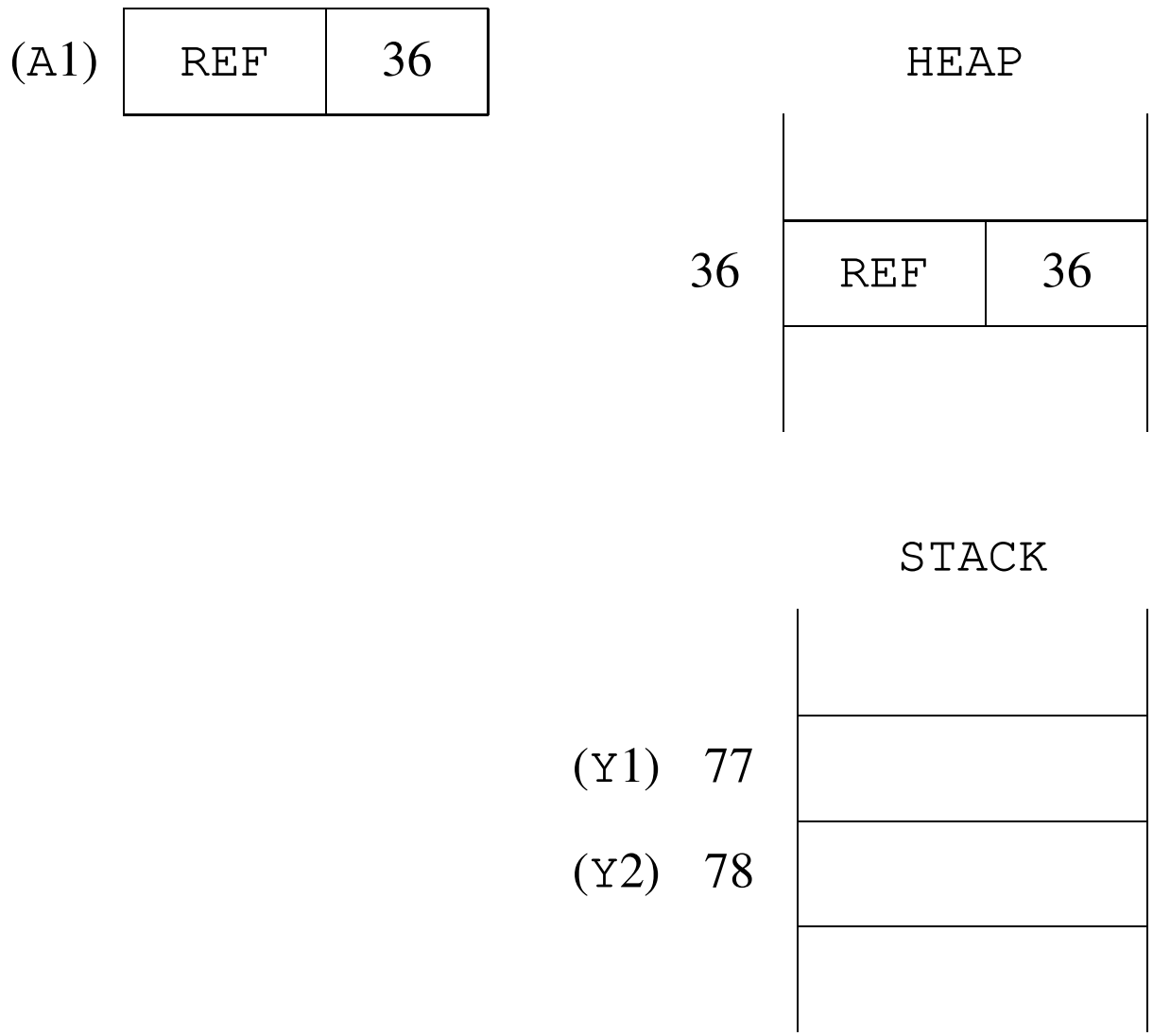


Unsafe code for $p(X) :- q(Y, X), r(Y, X).$

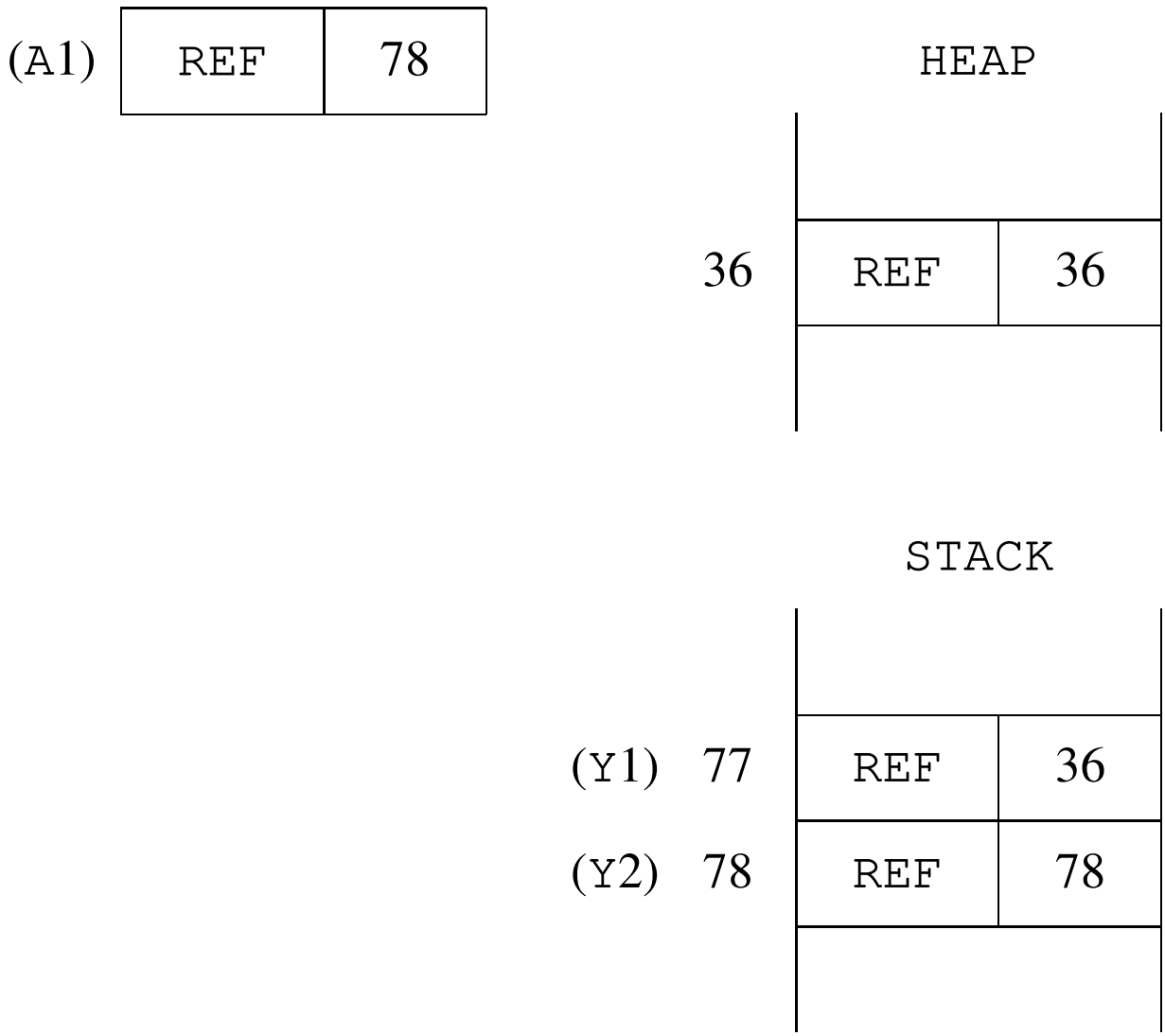
Before Line 0, `A1` points to the heap address (say, 36) of an unbound `REF` cell at the top of the heap:



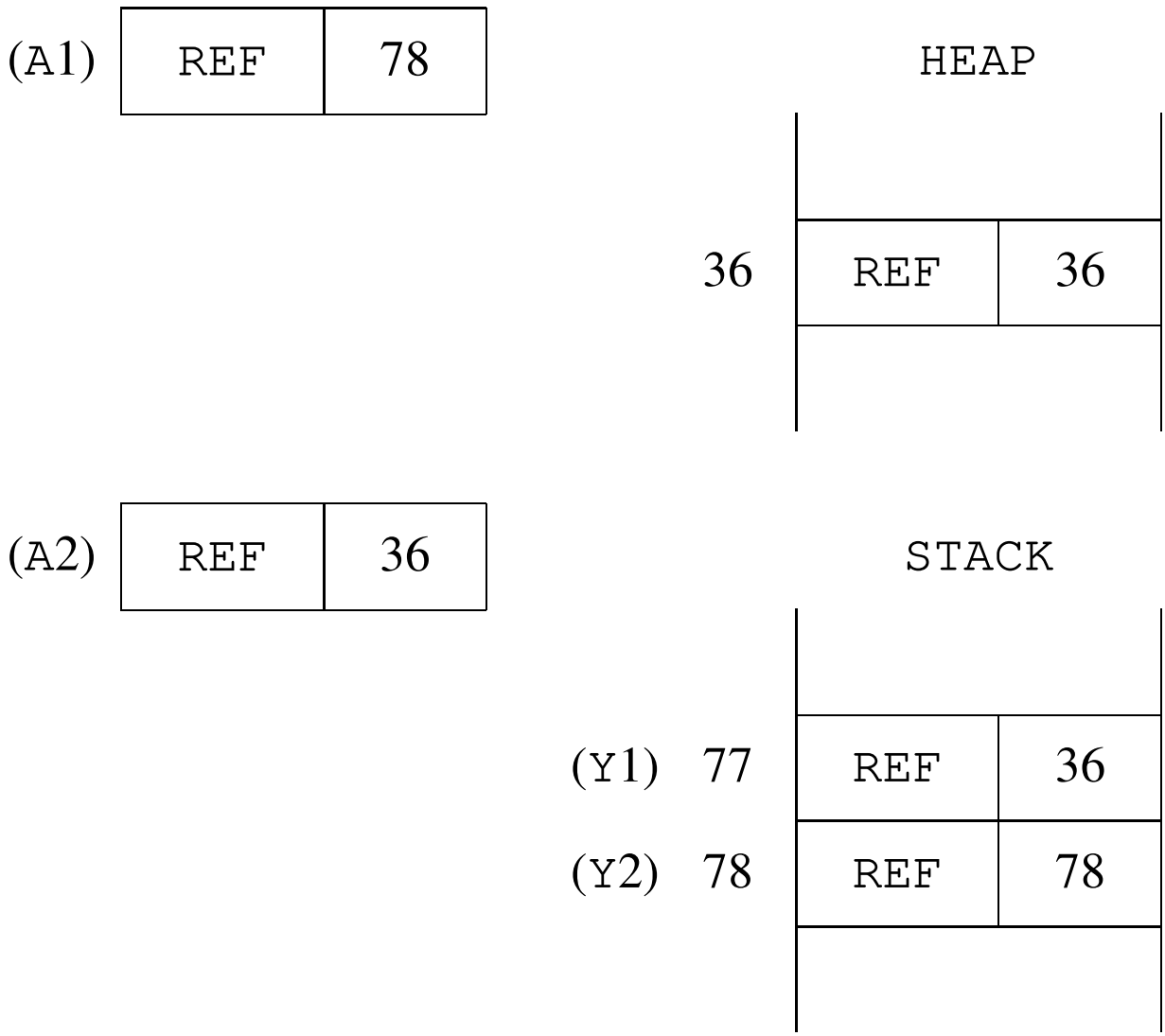
Then, `allocate` creates an environment on the stack (where, say, `Y1` is at address 77 and `Y2` at address 78 in the stack):



Line 1 sets `STACK[77]` to ⟨ `REF` , 36 ⟩, and Line 2 sets `A1` (and `STACK[78]`) to ⟨ `REF` , 78 ⟩.

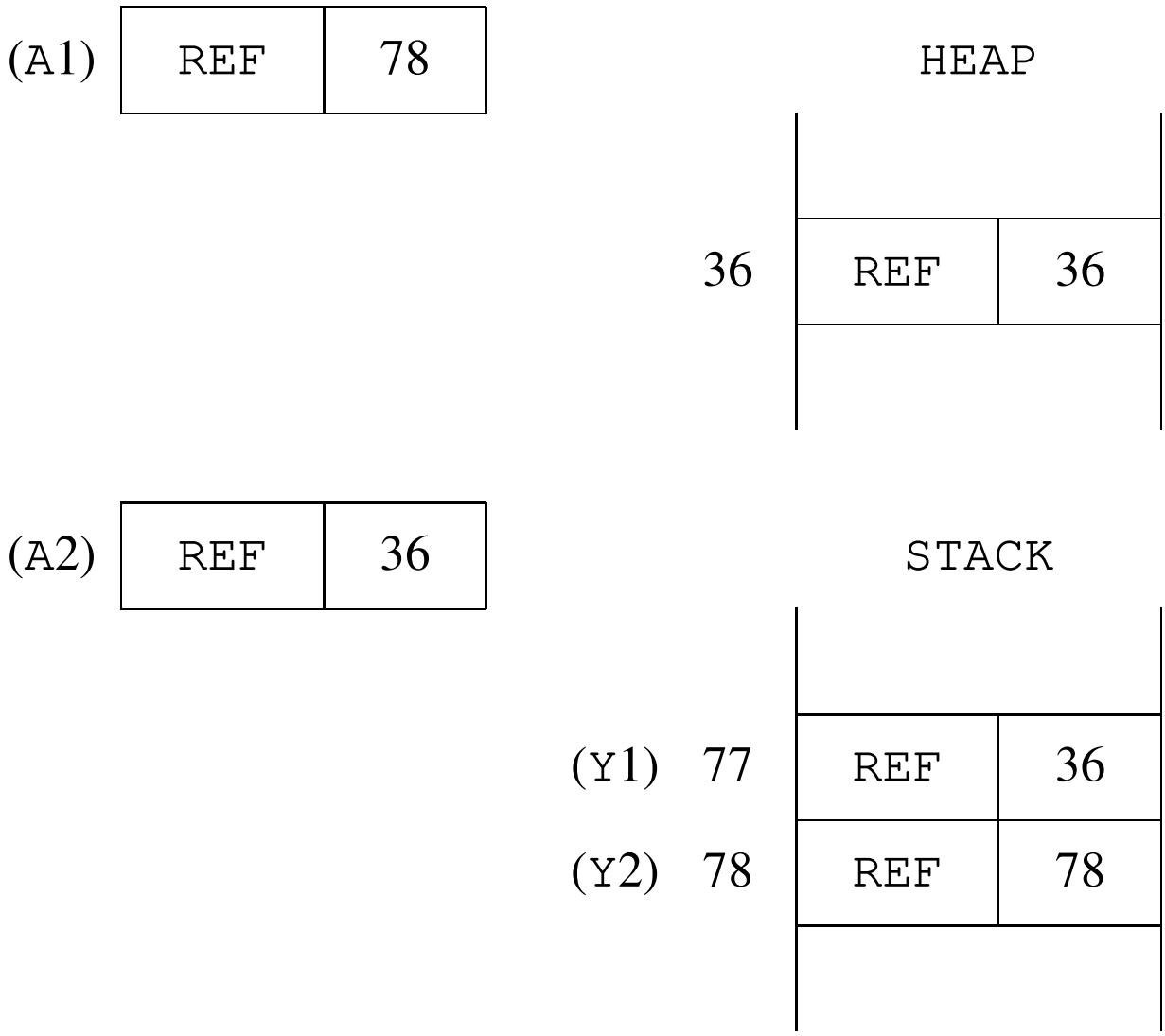


Line 3 sets A2 to the value of STACK[77]; that is, ⟨ REF , 36 ⟩.

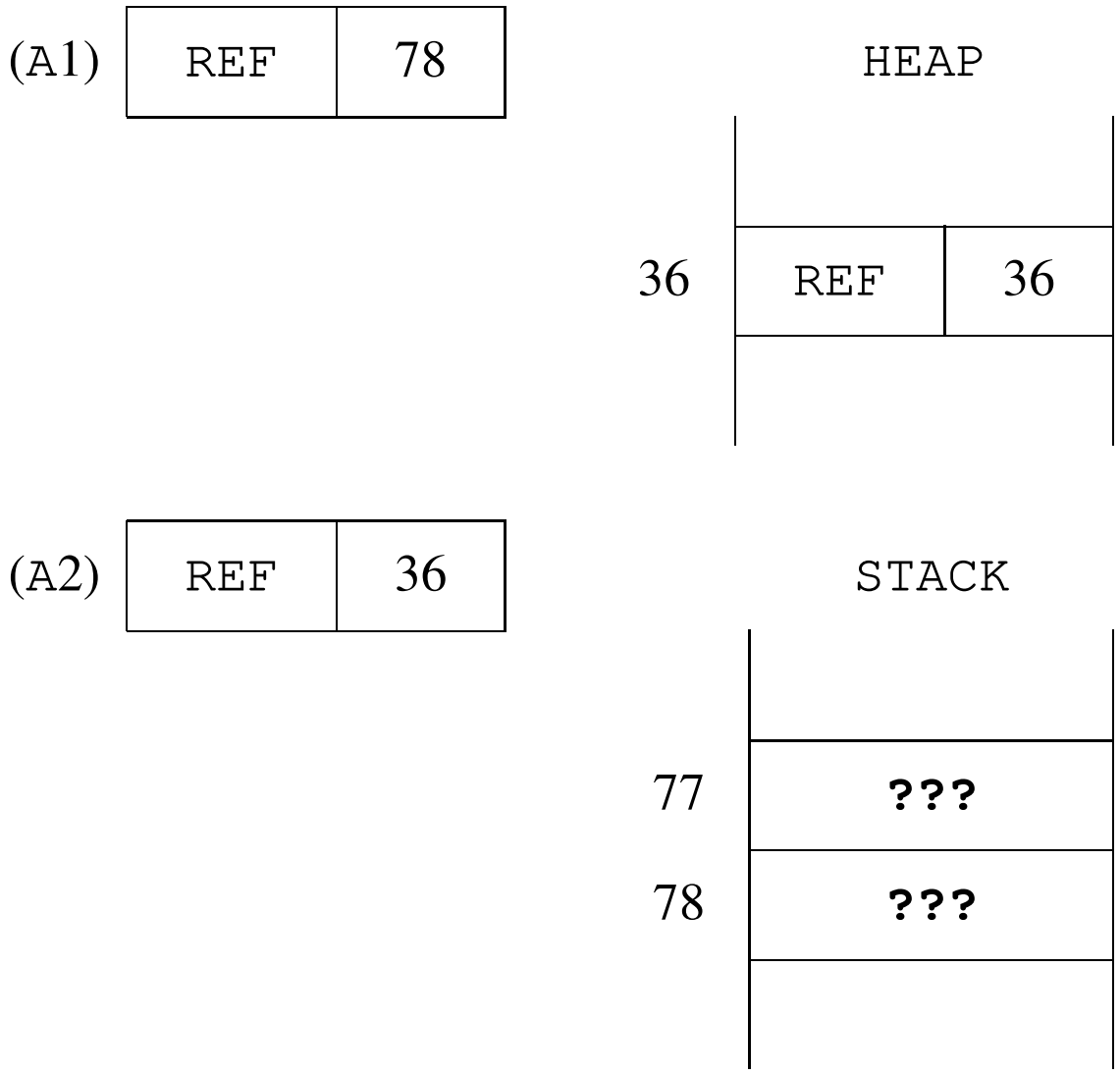


Assume now that the call to q on Line 4 does not affect these settings at all (*e.g.*, the fact $q(\_, \_)$ is defined).

Then, (the wrong) Line 5 would set A1 to $\langle$ REF , 78 $\rangle$, and Line 6 sets A2 to $\langle$ REF , 36 $\rangle$:



Next, `deallocate` throws away `STACK[77]` and `STACK[78]`.



LO! The code for r will find garbage in `A1`.

Remedy for unsafe variables

Two possible situations of an unsafe variable `Y`n in the last goal where it occurs:

- `Y`n appears only as an argument of its last goal;
- `Y`n appears in that goal nested in a structure, whether or not it is also an argument.

We defer the 2nd case: it is a more general source of unsafety that we shall treat later.

When all occurrences of unsafe `Y`n are arguments of the last goal where `Y`n appears, they all are `put_value Y`n, `A`i's.

Then, replace the *first* of its last goal's `put_value Y`n, `A`i's with `put_unsafe_value Y`n, `A`i.

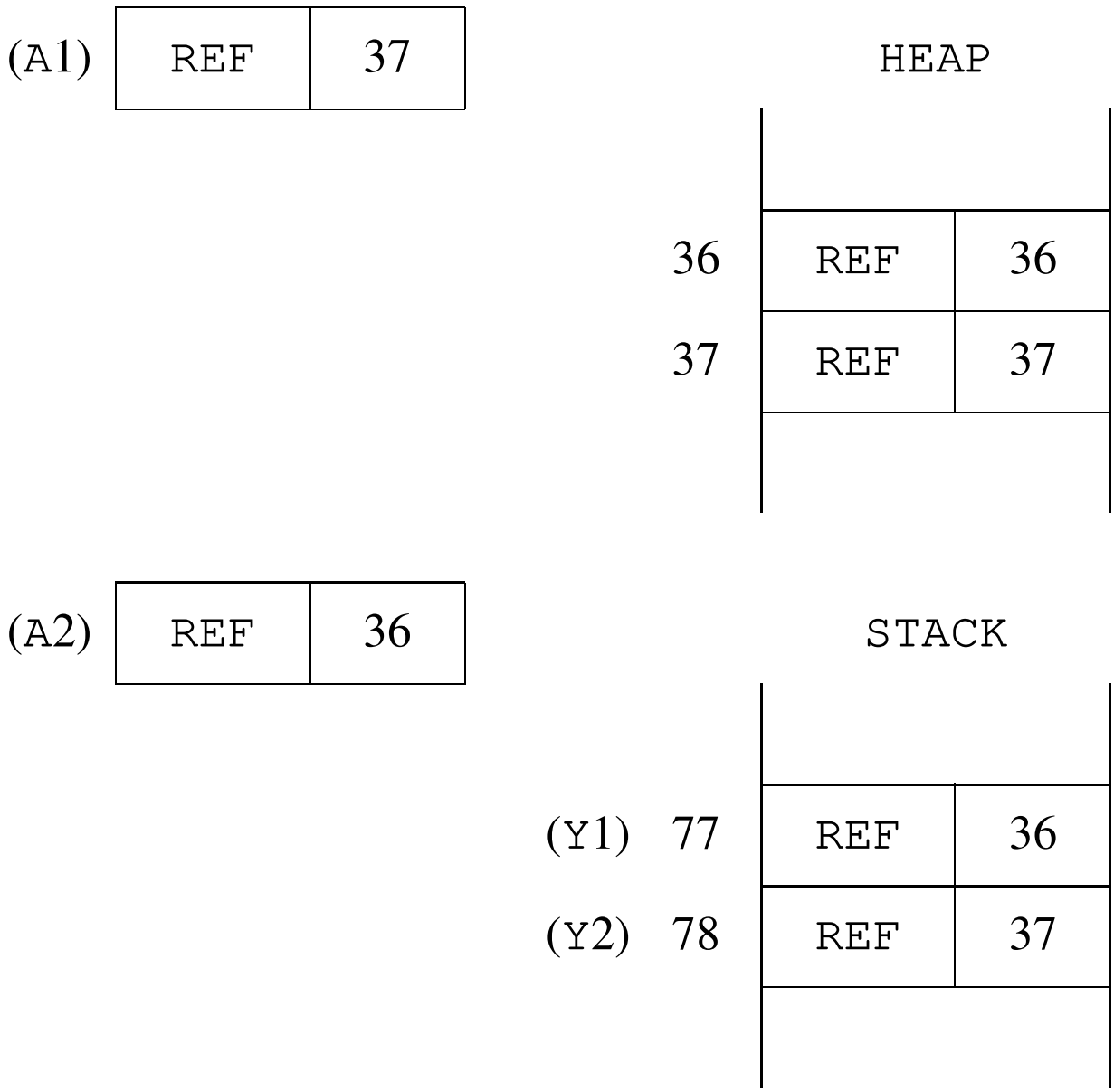
`put_unsafe_value` `Y`n, `A`i modifies `put_value` `Y`n, `A`i such that:

- if `Y`n does not lead to an unbound variable in the current environment, do `put_value` `Y`n, `A`i;
- otherwise, bind the stack variable to a new unbound `REF` cell on the heap, and set `A`i to it.

```
put_unsafe_value Yn, Ai ≡
        addr ← deref(E + n + 1);
        if addr < E
         then Ai ← STORE[addr]
         else
          begin
           HEAP[H] ← ⟨ REF , H ⟩;
           bind(addr, H);
           Ai ← HEAP[H];
           H ← H + 1
          end;
```

Back to example:

If Line 5 is `put_unsafe_value` Y2, A1, then `HEAP[37]` is created and set to ⟨ `REF` , 37 ⟩, `STACK[78]` and A1 are set to ⟨ `REF` , 37 ⟩, then A2 is set to ⟨ `REF` , 36 ⟩ (the value of `STACK[77]`):



Discarding `STACK[77]` and `STACK[78]` is now safe as executing r will get correct values from A1 and A2.

Nested stack references

When an unsafe PV occurs in its last goal nested in a structure (*i.e.*, as a `set_value` or a `unify_value`), the situation reflects a more general pathology which may also affect TV's.

e.g.,

Rule: $a(X)$ `:-` $b(f(X))$.

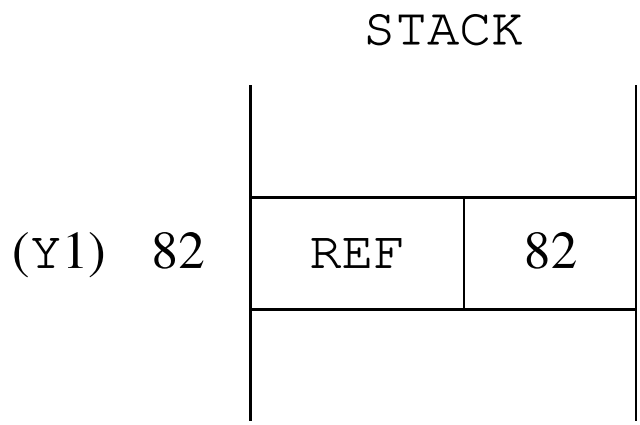
$a/1$: `get_variable` X2, A1
`put_structure` $f/1$, A1
`set_value` X2
`execute` $b/1$

Query: `?-`$a(X), \ldots$

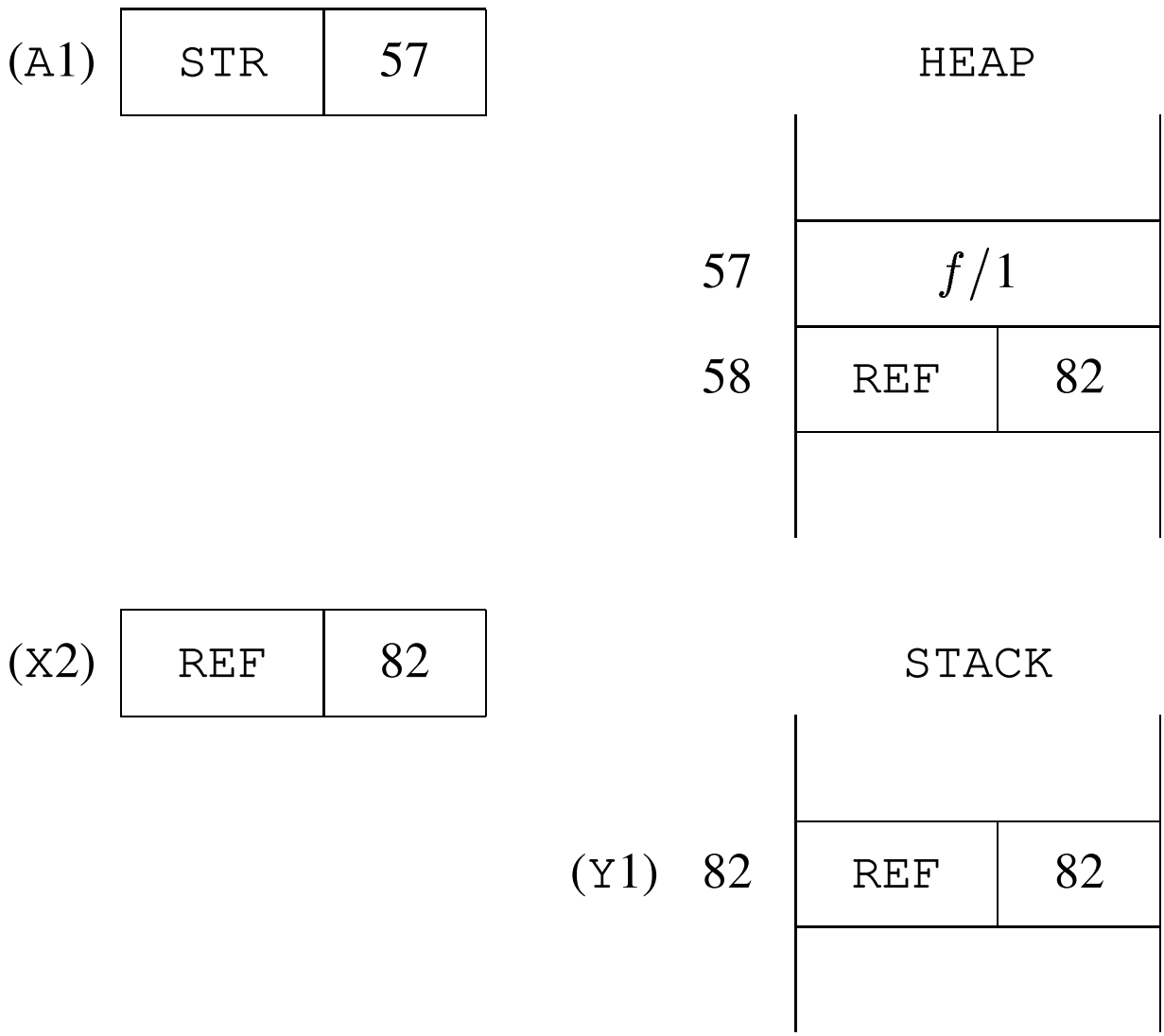
i.e.,

`allocate`
`put_variable` Y1, A1
`call` $a/1, 1$
⋮

Before the call to $a/1$, a stack frame containing `Y1` is allocated and initialized to unbound by `put_variable Y1,A1`:



Then X2 is set to point to that stack slot (the value of A1); functor $f/1$ is pushed on the heap; and `set_value` X2 pushes the value of X2 onto the heap:



Behold!, a reference from the heap to the stack.

This violates WAM Binding Rule 2 and creates a source of disaster when Y1 is eventually discarded.

Remedy for nested stack references

Question:

When can it be statically guaranteed that `set_value` (resp., `unify_value`) will not create an unwanted heap-to-stack reference?

Answer:

Any time its argument has not been explicitly initialized to be on the heap in the given clause.

i.e., `set_value` `V`n (resp., `unify_value` `V`n) is unsafe whenever the variable `V`n has *not* been initialized in this clause with `set_variable` or `unify_variable`, nor, if `V`n is temporary, with `put_variable`.

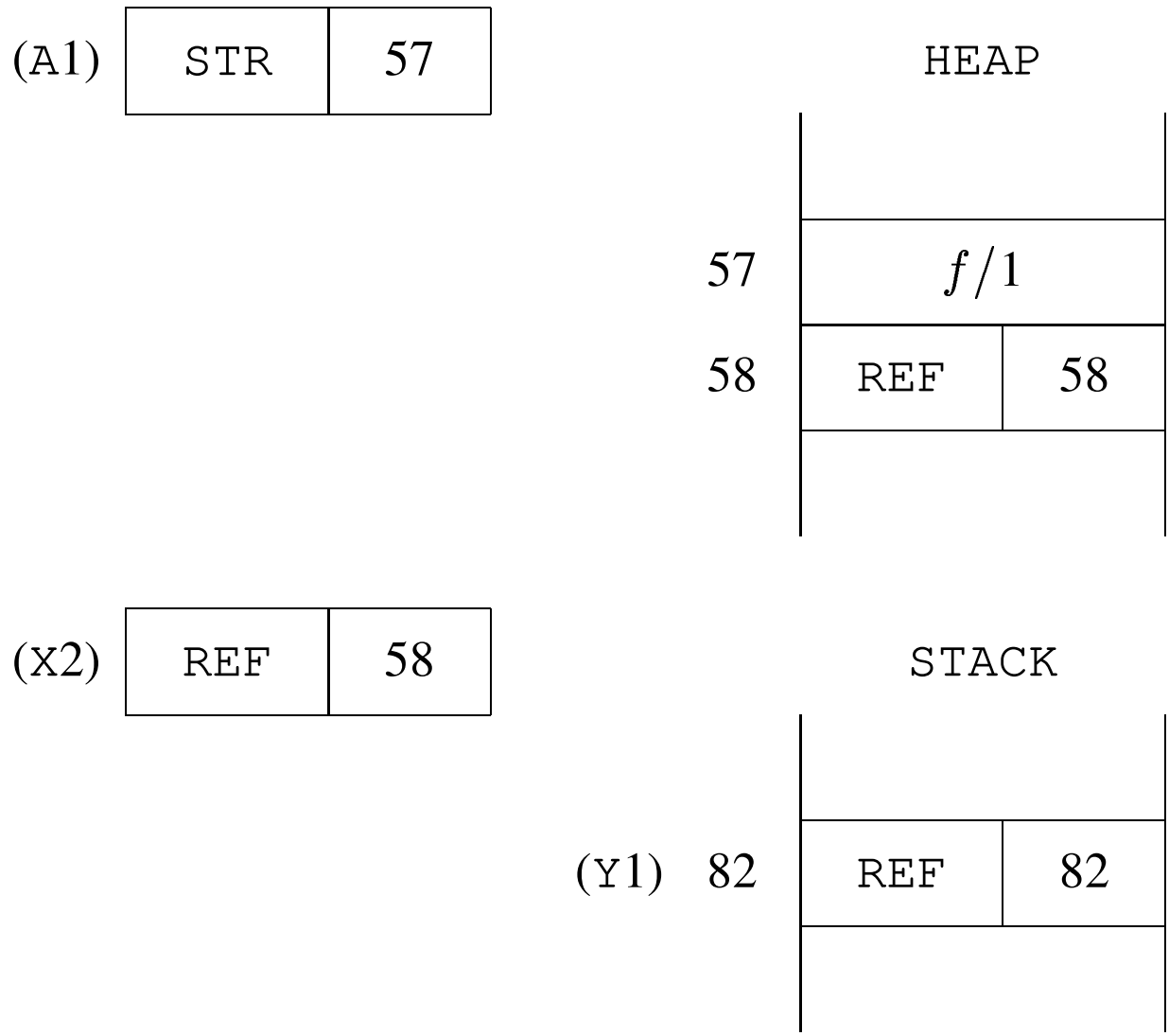
Cure:

Replace the *first* such `set_value` (resp., `unify_value`) with `set_local_value` (resp., `unify_local_value`.

```
set_local_value Vn ≡
        addr ← deref(Vn);
        if addr < H
         then HEAP[H] ← HEAP[addr]
         else
          begin
           HEAP[H] ← ⟨REF, H⟩;
           bind(addr, H)
          end;
        H ← H + 1;
```

Back to example:

If `set_local_value X2` replaces `set_value X2`, then it sees that the value of `X2` is a stack address and binds it to a new unbound cell on the heap.



This maintains a stack-to-heap reference, and WAM Binding Rule 2 is respected.

Variable classification revisited

NOTE: a PV is simply a conventional local variable (*i.e.*, allocated on the stack).

For David H. D. Warren,

- first, consider *all* variables as PV's;
- then, save stack space for those that are already initialized to previous data, are part of a structure existing on the heap, or must be globalized for LCO – call those TV's.

Warren's variable classification:

- A *temporary variable* is one which does not occur in more than one body goal (counting the head as part of the first body goal) *and* first occurs in the head, or in a structure, or in the last goal.
- A *permanent variable* is one which is not temporary.

NOTE:

- In both our and Warren's classification any variable occurring in more than one body goal is a PV;
- however, by Warren's (not ours) a PV may occur only in one body goal;

e.g., by our definition, X is a TV in:

$$a \ :- \ b(X, X), c.$$

but it is a PV by Warren's classification.

Problem: *Warren's variable classification is inconsistent with environment trimming, even with run-time safety checks.*

If X is a PV in:

$$a \ :- \ b(X, X), c.$$

then this compiles into:

```
a/0 : allocate                      %  a :-
      put_variable Y1, A1           %       b(X,
      put_unsafe_value Y1, A2       %           X
      call b/2, 0                   %             ),
      deallocate                    %       c
      execute c/0                   %         .
```

This is unsafe code:

- `Y1` is allocated on `STACK`;
- `A1` is set to the contents of `Y1`;
- `Y1` is found unsafe and must be globalized: set both `Y1` and `A2` to point to a new heap cell;
- `Y1` is discarded by ET;
- call $b/2$ with `A1` *still pointing to the discarded slot!*

Solution: *Delay ET for such PV's until following call*.

```
a/0 : allocate                  %  a :-
      put_variable Y1, A1       %       b(X,
      put_value Y1, A2          %            X
      call b/2, 1               %              ),
      deallocate                %       c
      execute c/0               %         .
```

Delayed trimming for a `:-` $b(X, X), c.$

i.e., `Y1` is kept in the environment until the time when execution returns from $b/2$, at which point it is discarded.

Indexing

To seed up clause selection, the WAM uses the first argument as indexing key.

NOTE: In a procedure's definition, a clause whose head has a variable key creates a search bottleneck.

$\Rightarrow$ A procedure p defined by the sequence of clauses

$$C_1, \ldots, C_n$$

is partitioned as a sequence of subsequences

$$S_1, \ldots, S_m$$

where each S_i is

- either a *single* clause with a variable key;
- or a *maximal* subsequence of contiguous clauses whose keys are not variables.

$$S_1 \begin{cases} call(X or Y) \ :- \ call(X). \\ call(trace) \ :- \ trace. \\ call(X or Y) \ :- \ call(Y). \\ call(notrace) \ :- \ notrace. \\ call(nl) \ :- \ nl. \end{cases}$$

$$S_2 \left\{ call(X) \ :- \ builtin(X). \right.$$

$$S_3 \left\{ call(X) \ :- \ extern(X). \right.$$

$$S_4 \begin{cases} call(call(X)) \ :- \ call(X). \\ call(repeat). \\ call(repeat) \ :- \ call(repeat). \\ call(true). \end{cases}$$

Compiling scheme for procedure p with definition partitioned into $S_1, \ldots, \mathbf{s}_m$, where $m > 1$:

$$
\begin{array}{lll}
p & : & \texttt{try\_me\_else}\ S_2 \\
 & & \textit{code for subsequence } S_1 \\
S_2 & : & \texttt{retry\_me\_else}\ S_3 \\
 & & \textit{code for subsequence } S_2 \\
 & & \vdots \\
S_m & : & \texttt{trust\_me} \\
 & & \textit{code for subsequence } S_m
\end{array}
$$

where `retry_me_else` is necessary only if $m > 2$.

If $m = 1$, none of the above is needed and the translation boils down only to the code necessary for the single subsequence chunk.

For a degenerate subsequence (*i.e.*, single variable-key clause) translation is as usual.

```
call/1 : try_me_else S2      %
         indexed code for S1 %
S2     : retry_me_else S3    %  call(X)
         execute builtin/1   %          :- builtin(X).
S3     : retry_me_else S4    %  call(X)
         execute extern/1    %          :- extern(X).
S4     : trust_me            %
         indexed code for S4 %
```

Indexing a non-degenerate subsequence

General indexing code pattern:

first level indexing;
second level indexing;
third level indexing;
code of clauses in subsequence order;

where:

- second and third levels are needed only depending on what sort of keys are present in the subsequence and in what number;
- they disappear in the degenerate cases;
- following dispatching code is the regular sequential choice control construction.

First level dispatching makes control jump to a (possibly void) bucket of clauses, depending on whether $deref(\mathtt{A1})$ is:

- *a variable;*

  the code bucket of a variable corresponds to full sequential search through the subsequence (thus, it is never void);

- *a constant;*

  the code bucket of a constant corresponds to second level dispatching among constants;

- *a (non-empty) list;*

  the code bucket of a list corresponds:

  – either to the single clause with a list key,

  – or to a linked list of all those clauses in the subsequence whose keys are lists;

- *a structure;*

  the code bucket of a structure corresponds to second level dispatching among structures;

For those constants (or structures) having multiple clauses, a possible third level bucket corresponds to the linked list of these clauses (just like the second level for lists).

first level indexing for S_1
second level indexing for S_1
third level indexing for S_1

S_{11} : `try_me_else` S_{12}
code for '$call(XorY)$:- $call(X)$.'

S_{12} : `retry_me_else` S_{13}
code for '$call(trace)$:- $trace$.'

S_{13} : `retry_me_else` S_{14}
code for '$call(XorY)$:- $call(Y)$.'

S_{14} : `retry_me_else` S_{15}
code for '$call(notrace)$:- $notrace$.'

S_{15} : `trust_me`
code for '$call(nl)$:- nl.'

Indexing instructions

First level dispatching:

- `switch_on_term` V, C, L, S

  jump to the instruction labeled V, C, L, or S, depending on whether $deref(\texttt{A1})$ is, respectively, a variable, a constant, a non-empty list, or a structure.

Second level dispatching: for N distinct symbols,

- `switch_on_constant` N, T

  (T is a hash-table of the form $\{c_i : L_{c_i}\}_{i=1}^{N}$)

  if $deref(\texttt{A1}) = c_i$, jump to instruction labeled L_{c_i}. Otherwise, backtrack.

- `switch_on_structure` N, T

  (T is a hash-table of the form $\{s_i : L_{s_i}\}_{i=1}^{N}$)

  if $deref(\texttt{A1}) = s_i$, jump to instruction labeled L_{s_i}. Otherwise, backtrack.

Third level indexing:

Thread together a sequence of multiple (not necessarily contiguous) clauses whose keys are lists, or a same constant or structure, using:

- `try` L,
- `retry` L,
- `trust` L.

They are identical to `try_me_else` L, `retry_me_else` L, and `trust_me`, respectively, except that they jump to label L and save the next instruction in sequence as the next clause alternative in the choice point (except for `trust`, of course).

NOTE: Second level for lists is really third level indexing on list structures, the second level being skipped by special handling of lists in the spirit of WAM Principle 3.

```
        switch_on_term S11, C1, fail, F1                    % 1st level dispatch for S1
C1  :   switch_on_constant 3, { trace    : S1b,             % 2nd level for constants
                                notrace  : S1d,
                                nl       : S1e }
F1  :   switch_on_structure 1, { or/2 : F11 }               % 2nd level for structures
F11 :   try S1a                                             % 3rd level for or/2
        trust S1c                                           %
S11 :   try_me_else S12                                     % call
S1a :   get_structure or/2, A1                              %     (or
        unify_variable A1                                   %        (X,
        unify_void 1                                        %           Y))
        execute call/1                                      %     :- call(X).
S12 :   retry_me_else S13                                   % call
S1b :   get_constant trace, A1                              %     (trace)
        execute trace/0                                     %     :- trace.
S13 :   retry_me_else S14                                   % call
S1c :   get_structure or/2, A1                              %     (or
        unify_void 1                                        %        (X,
        unify_variable A1                                   %           Y))
        execute call/1                                      %     :- call(Y).
S14 :   retry_me_else S15                                   % call
S1d :   get_constant notrace, A1                            %     (notrace)
        execute notrace/0                                   %     :- notrace.
S15 :   trust_me                                            % call
S1e :   get_constant nl, A1                                 %     (nl)
        execute nl/0                                        %     :- nl.
```

Indexing code for subsequence S_1

```
S4     switch_on_term S41, C4, fail, F4          % 1st level dispatch for S4
C4   : switch_on_constant 3, { repeat : C41,     % 2nd level for constants
                               true   : S4d }
F4   : switch_on_structure 1, { call/1 : S41 }   % 2nd level for structures
C41  : try S4b                                   % 3rd level for 'repeat'
       trust S4c                                 %
S41  : try_me_else S42                           % call
S4a  : get_structure call/1, A1                  %      (call
       unify_variable A1                         %           (X))
       execute call/1                            %       :- call(X).
S42  : retry_me_else S43                         % call
S4b  : get_constant repeat, A1                   %      (repeat)
       proceed                                   %                .
S43  : retry_me_else S44                         % call
S4c  : get_constant repeat, A1                   %      (repeat)
       put_constant repeat, A1                   %       :- call(repeat)
       execute call/1                            %                       .
S44  : trust_me                                  % call
S4d  : get_constant true, A1                     %      (true)
       proceed                                   %               .
```

Indexing code for subsequence S_4

$conc([], L, L).$
$conc([H|T], L, [H|R]) \texttt{ :- } conc(T, L, R).$

$conc/3$	: `switch_on_term` $C_{1a}, C_1, C_2, fail$	%		
C_{1a}	: `try_me_else` C_{2a}	%	$conc$	
C_1	: `get_constant` [], A1	%	([],	
	`get_value` A2, A3	%	$L, L)$	
	`proceed`	%	.	
C_{2a}	: `trust_me`	%	$conc$	
C_2	: `get_list` A1	%	([	
	`unify_variable` X4	%	$H\|$	
	`unify_variable` A1	%	$T], L,$	
	`get_list` A3	%	[	
	`unify_value` X4	%	$H\|$	
	`unify_variable` A3	%	$R])$	
	`execute` $conc/3$	%	$\texttt{ :- } conc(T, L, R).$	

Encoding of $conc/3$

NOTE: When $conc/3$ is called with an instantiated first argument, no choice point frame for it is ever needed.

In fact, incidentally to achieving faster search, indexing has major serendipitous benefits:

- it *substantially* reduces the creation and manipulation of choice point frames;
- it eliminates useless environment protection;
- it magnifies the effect of LCO and ET.

Cut

! : succeed and forget any other potential alternative for this procedure as well as any other arising from preceding body goals.

i.e., discard all choice points created after the choice point that was current right before calling this procedure.

Backtrack Cut Register: **BC**

BC keeps the choice point where to return upon backtracking over a cut.

BC must contain the address of the choice point that is current at the time a procedure call is made:

⇒ alter `call` and `execute` to set **BC** to the value of the current value of **B**;

⇒ cut amounts to resetting **B** to the value of **BC**.

(NOTE: **BC** must be saved as part of a choice point, and and restored upon backtracking.)

Two sorts of cuts:

- *shallow* (or *neck*) cuts; *e.g.*,

$$h \texttt{ :- } \texttt{!}, b_1, \ldots, b_n.$$

- *deep* cuts; *e.g.*,

$$h \texttt{ :- } \ldots, b_i, \texttt{!}, \ldots, b_n. \ (1 \leq i \leq n).$$

Neck cut

- `neck_cut`

  discard any (one or two) choice points following **B** (*i.e.*, **B** $\leftarrow$ **BC**, **HB** $\leftarrow$ **B**.**H**).

e.g.,

$$a \texttt{ :- } \texttt{!}, b.$$

is compiled into:

```
neck_cut
execute b/0
```

Deep cut

- `get_level` `Y`n

  immediately after `allocate`, set `Y`n to current **BC**;

- `cut` `Y`n

  discard all (if any) choice points after that indicated by `Y`n, and eliminate new unconditional bindings from the trail up to that point.

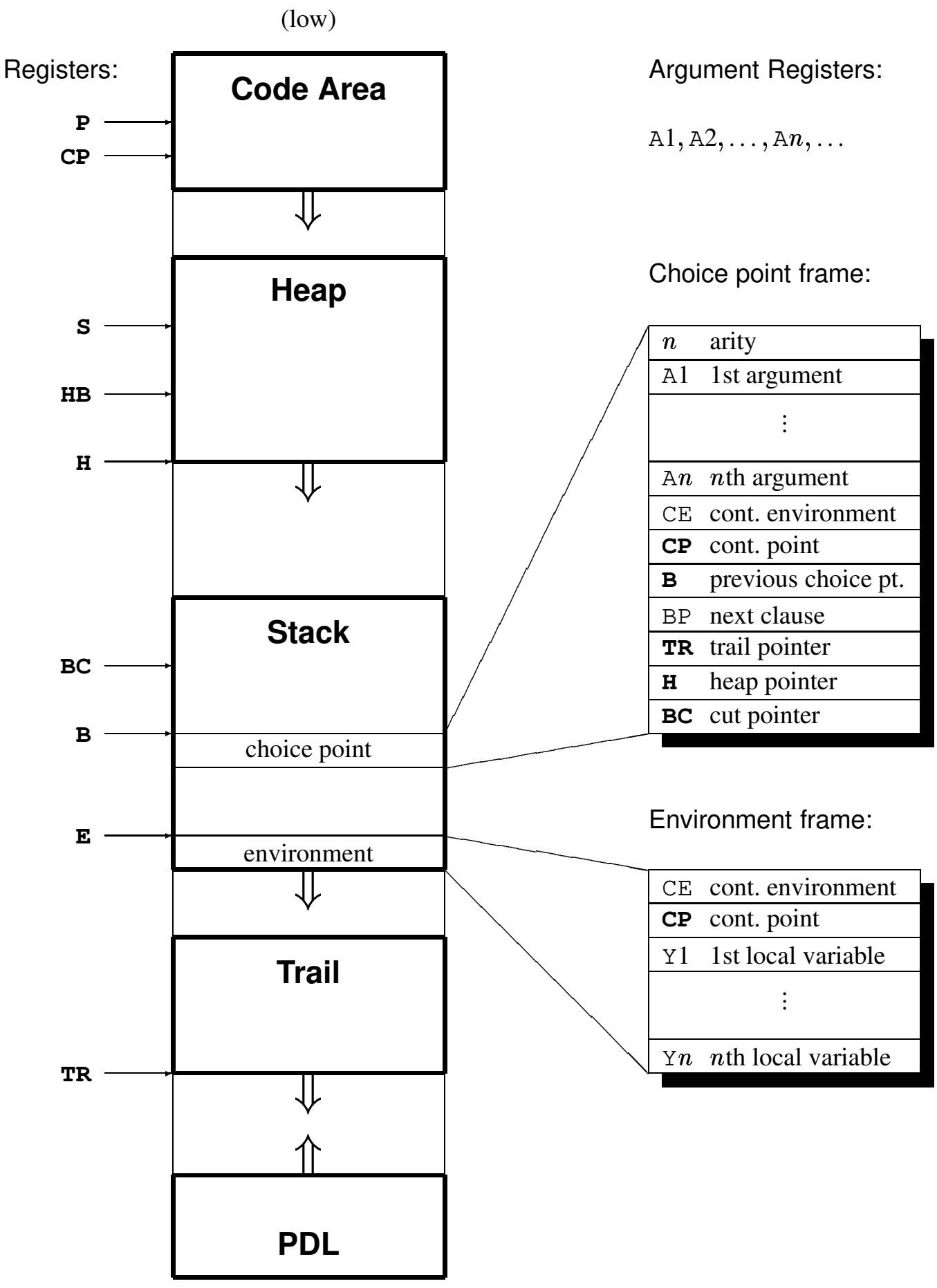
e.g.,

$$a \texttt{ :- } b, \texttt{!}, c.$$

is compiled into:

```
allocate
get_level Y1
call b/0,1
cut Y1
deallocate
execute c/0
```

WAM Memory Layout and Registers



The Complete WAM Instruction Set

Put instructions

`put_variable` Xn, Ai
`put_variable` Yn, Ai
`put_value` Vn, Ai
`put_unsafe_value` Yn, Ai
`put_structure` f, Ai
`put_list` Ai
`put_constant` c, Ai

Get instructions

`get_variable` Vn, Ai
`get_value` Vn, Ai
`get_structure` f, Ai
`get_list` Ai
`get_constant` c, Ai

Set instructions

`set_variable` Vn
`set_value` Vn
`set_local_value` Vn
`set_constant` c
`set_void` n

Unify instructions

`unify_variable` Vn
`unify_value` Vn
`unify_local_value` Vn
`unify_constant` c
`unify_void` n

Control instructions

`allocate`
`deallocate`
`call` P, N
`execute` P
`proceed`

Choice instructions

`try_me_else` L
`retry_me_else` L
`trust_me`
`try` L
`retry` L
`trust` L

Indexing instructions

`switch_on_term` V, C, L, S
`switch_on_constant` N, T
`switch_on_structure` N, T

Cut instructions

`neck_cut`
`get_level` Yn
`cut` Yn

NOTE: In some instructions, we use the notation Vn to denote a variable that may be indifferently temporary or permanent.

Bibliography

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, Logic Programming Series. Cambridge, MA, 1991.

[2] Saumya K. Debray. Register allocation in a Prolog machine. In *Proceedings of the Symposium on Logic Programming*, pages 267–275. IEEE Computer Society, September 1986.

[3] Peter Kursawe. How to invent a Prolog machine. *New Generation Computing*, 5:97–114, 1987.

[5] David M. Russinoff. A verified Prolog compiler for the Warren abstract machine. MCC Technical Report Number ACT-ST-292-89, Microelectronics and Computer Technology Corporation, Austin, TX, July 1989.

[6] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[7] David H. D. Warren. Implementation of Prolog. Lecture notes, Tutorial No. 3, 5th International Conference and Symposium on Logic Programming, Seattle, WA, August 1988.