Computational Logic

Introduction to Prolog Implementation:
The Warren Abstract Machine
(WAM)

(Text derived from the tutorial at the
1989 International Conference
on Logic Programming)

Evolution of the WAM:

1974 Marseille	**Battani-Meloni** **Prolog**	Interpreter in Fortran	Structure-sharing
	↓		
1977 Edinburgh	**DEC-10 Prolog** ↓	Compiler to native code	Structure-sharing, multiple stacks: recovery of storage
	↓→ Icot Machine (PSI)		on det. ret., TRO, cut
	↓		
	Portable Prolog Compiler [Bowen et. al] ...		
	↓		
1983 SRI	**"Old Engine"** ↓ ↓ ↓	compiler to abstract machine code + emulator	structure copying, goal stacking
1983/4 SRI	"New Engine" (**WAM**) ↓	compiler to abstract machine code + emulator	structure copying, environment stacking, env. trimming,

↓ → SW →   Quintus, SICStus, BIM, ALS, LPA, etc.
↓ → HW →   Tick/Warren "overlapped Prolog processor,"
↓   Berkeley PLM, NEC HPM, ECRC, etc.
↓ → Multiprocessor implementations (RAP-WAM, SRI, ...).
...

WAM [Warren 83]: A series of compilation techniques and run-time algorithms which attain high execution speed and storage efficiency.

Format: abstract machine, i.e. instruction set + storage model.

[Hogger 84, Maier & D.S. Warren 88, Ait-Kaci 90]
"Up to and including the WAM"

Fundamental Operations:

Procedure control

- calling procedures
- allocating storage
- returning
- tail (last call) recursion

Parameter passing / unification

- unification (customized)
- loading and unloading of parameter registers
- variable classification
- variable binding / trailing

Choice points, failure, backtracking

- creation, update, and deletion of choice points
- recovery of space on backtracking
- unbinding of variables

Indexing

- on parameter type (tag = var, struct, const, list...)
- on principal functor / constant (hash table)

Other

- cut
- arithmetic
- etc.

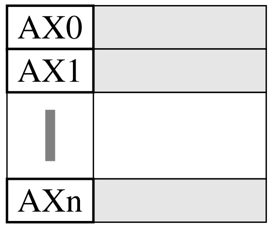
Functions performed and elements performing them:

¤ Parameter Passing:

° Through **argument registers**

..., f(a), ...

```
..
put_constant  a,X1
call  f/1, ...
```



allows register allocation optimizations

¤ Unification:

° "Customization" (open coding)

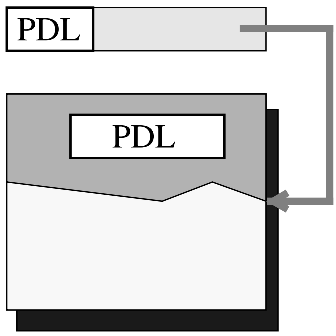
° **push-down list** (PDL)

f(x) :- ...

```
get_var  Y1,X1
...
```

f(a) :- ...

```
get_constant  a,X1
...
```



Functions performed and elements performing them:

¤ Code Storage and Sequencing:

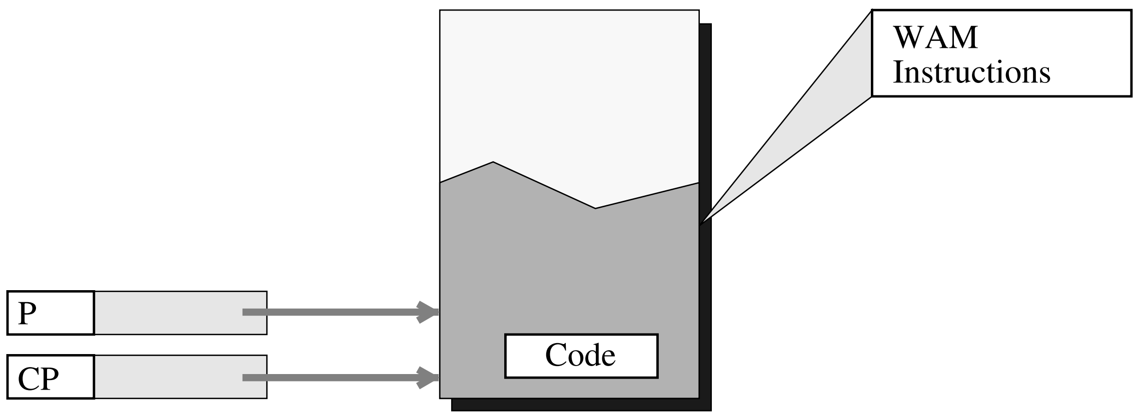
- **Code Space** (a stack/heap)
- **P**: Program Counter
- **CP**: Continuation Pointer

..., f(a), ...

```
...
put_constant  a,X1
call  f/1,...
```

f(a).

```
get_constant  a,X1
proceed
```



Functions performed and elements performing them:

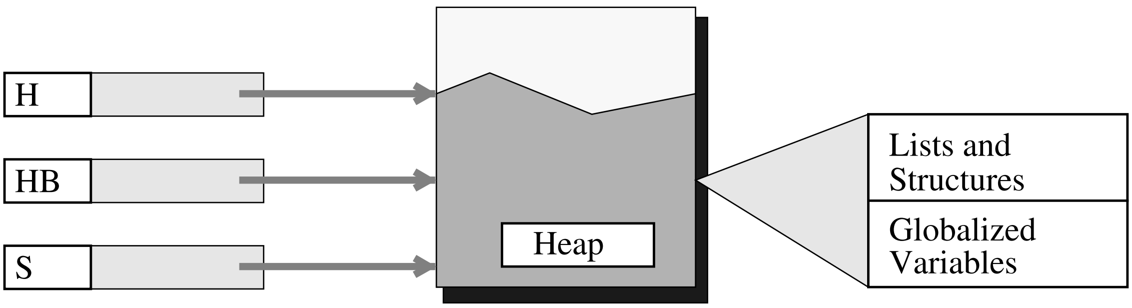
¤ Global Data Storage:

- The **Heap** (a stack/heap). Contains lists, structures, and global variables.

  ° **H**: Top of Heap

  ° **HB**: Heap Backtrack pointer

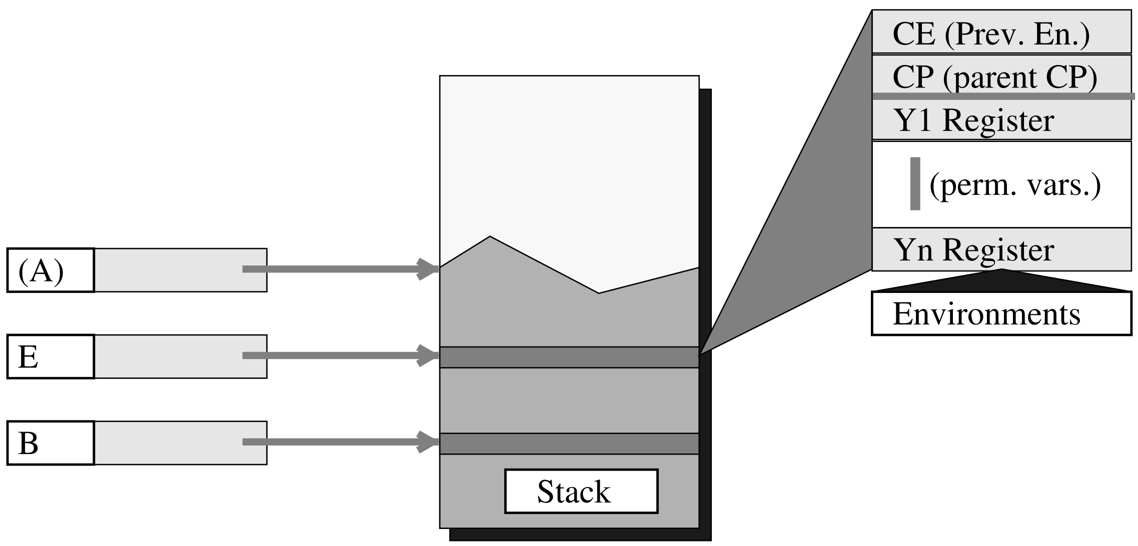
  ° **S**: Structure Pointer (Read Mode)



Functions performed and elements performing them:

¤ Local data storage + control (forward execution):

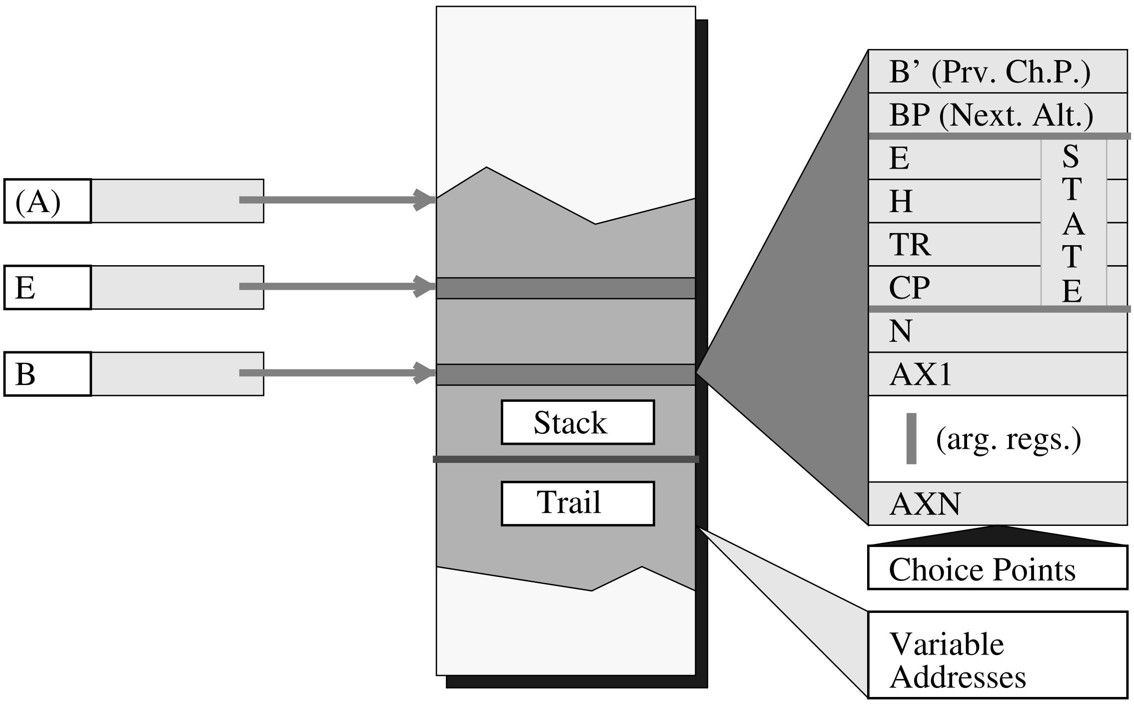
- The **Stack** (a stack/heap). Contains ***environments*** and ***choice points***.
  - **A**: Top of Stack (not required)
  - **B**: Choice Point pointer
  - **E**: Environment pointer
- **Environments**:
  - Permanent (local) variables
  - Control information



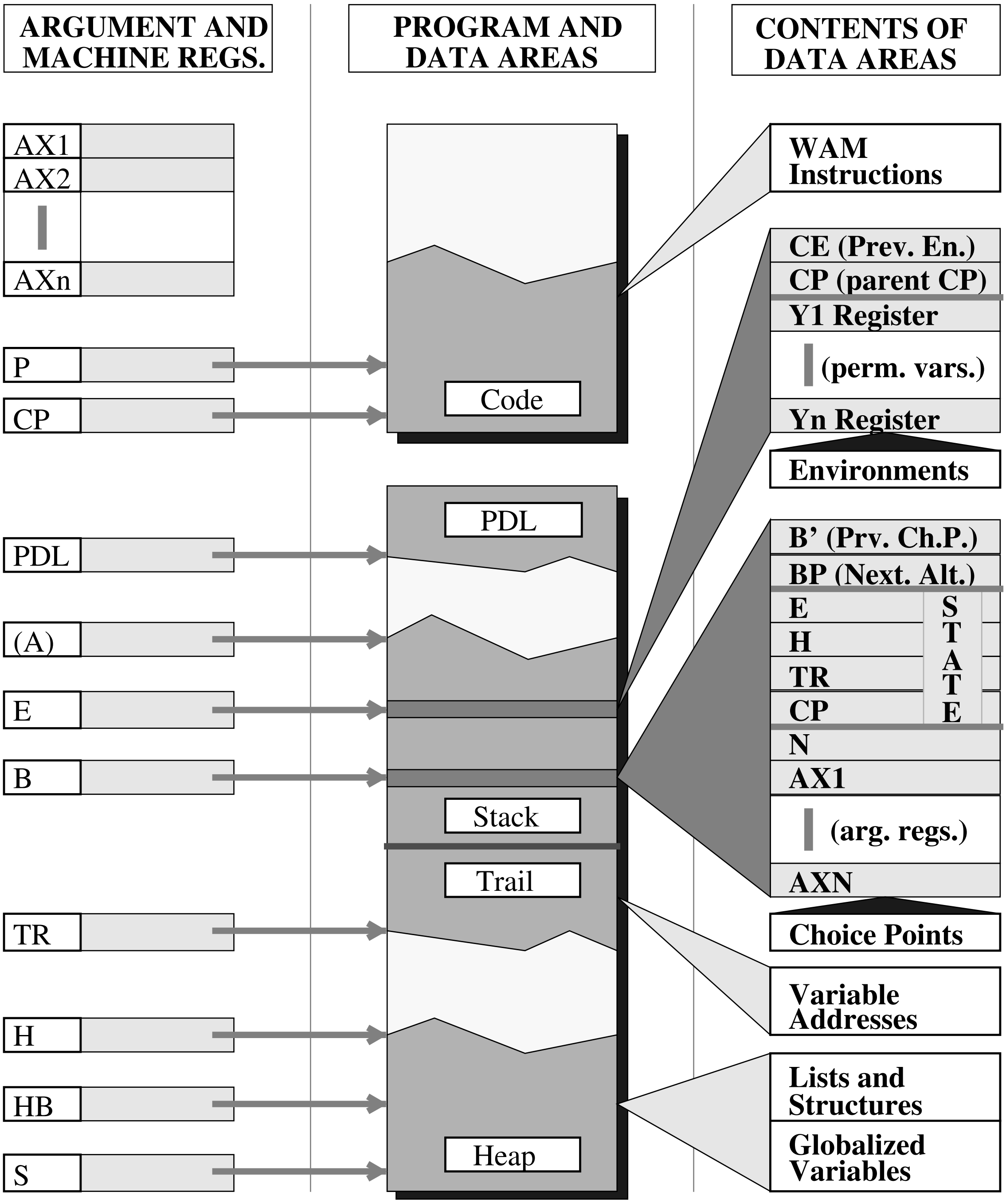
Functions performed and elements performing them:

¤ Control (backtracking):

- **Choice Points**: reside in the Stack.
  - ° State of the machine at the time of entering an alternative
  - ° Pointer to next alternative
- The **Trail**:
  - ° Addresses of variables which need to be unbound during backtracking.

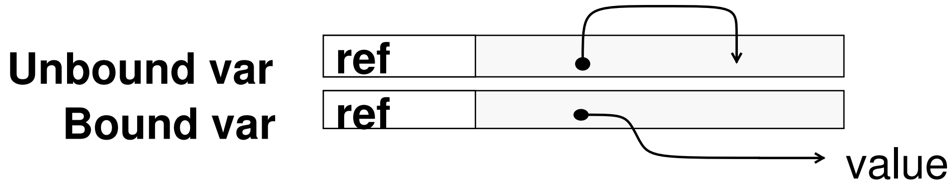


WAM Storage Model



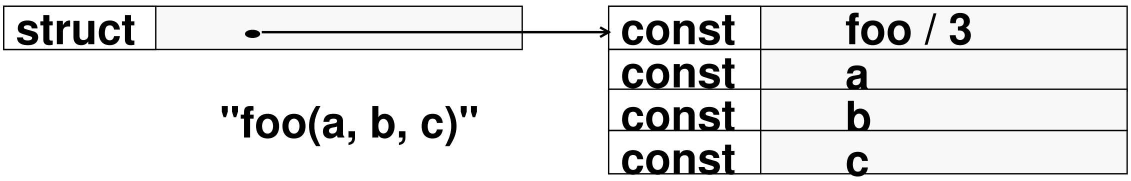
Data Types:

1.- Reference: represents variables.

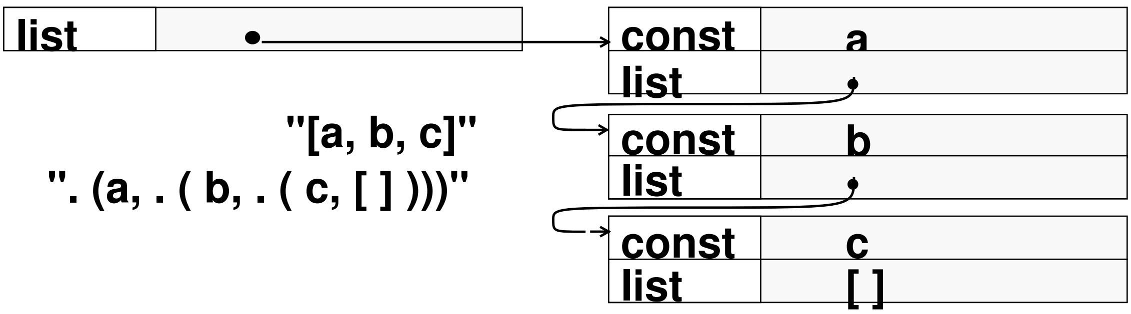


2.- Constant: represents atoms, ints., ..

3.- Structure: represents structures (other than lists).



4.- List: special case of structure.



Variable Classification:

- ***Permanent Variables:*** those which need to "survive" across procedure calls. They live in the Stack ("Y" registers in the environment).
- ***Temporary Variables:*** all others, they are allocated in the real registers ("AX" registers).
- ***Global Variables:*** those which need to survive the environment. They live in the Heap.

Permanent and *Temporary* variables correspond to the traditional concept of ***local*** variables.

grandparent(X, Y):- parent(X, Z), parent(Z, Y).

temporary

permanent

global ("unsafe")

Variable Binding and Dereferencing:

1.- Binding a variable to a non-variable:

- Overwrite (trail if necessary).

2.- Binding a variable to another variable:

- Bind so that younger variables point to older variables
- Bind at end of dereferencing chain
- Variables in the Stack should point to the Heap (not otherwise).

Accomplished with a simple address comparison (if data areas arranged correctly in memory).

Trailing:

Store in the Trail the address of a variable which is being bound only if it is

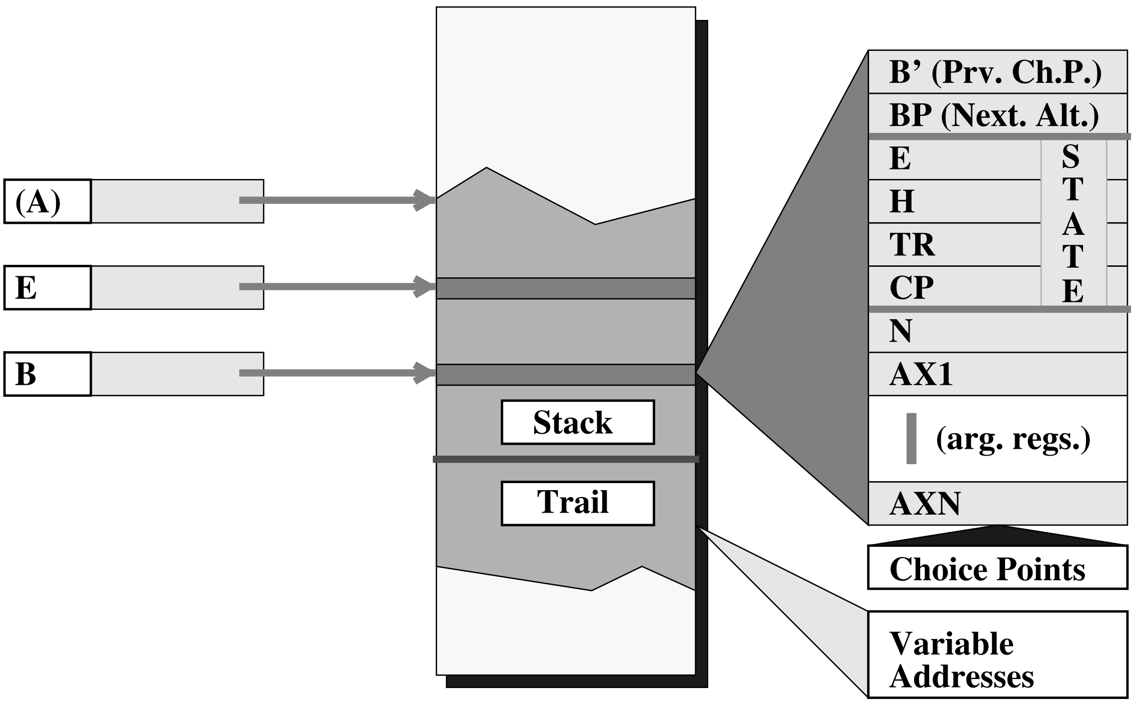
- Before HB if in the Heap
- Before B if in the Stack

Failure: (at "get," "unify," ...)

1.- Restore registers from current choice-point (machine and AX registers)

2.- Get TR from Choice Point. Pop addresses from Trail until TR. Set all these variables to "unbound" (fast)

3.- Begin execution of the next alternative at BP



Unification Modes:

¤ Unification can perform two tasks (during execution of "unify" instructions):

- Pattern matching ⟶ ***READ mode***
- Term construction ⟶ WRITE mode

The decision is made dynamically: "append"

```
append([X|L1], L2, [X|L3]) :- append(L1,L2,L3).

get_list          A1       % [
unify_variable    X4       %   X |
unify_variable    A1       %       L1], L2,
get_list          A3       %              [
...                        %               ...
```

READ mode: X4 := next arg. (from S); (S++)
WRITE mode: X4 := ref to next arg (from H), which is initialized to "unbound"; (H++)

The same code for "append" has to do both tasks: ***READ*** and WRITE.

Mode must be preserved across instructions.

Last Call Optimization:

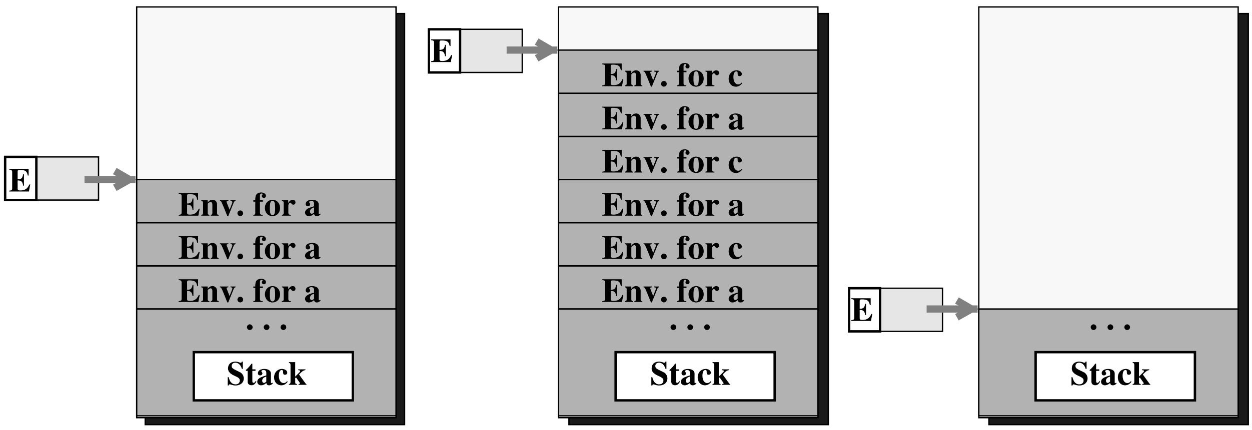
An extension of tail recursion optimization:

- All storage local to a clause (i.e. the environment) is deallocated *prior to calling the last goal in the body.*
- Turns tail recursions and last call mutual recursions into real iteration: the stack doesn't grow.

Example:

```
            ?:- a(3).

            a(0).
(1) ⟶      a(N) :- b, c, NN is N-1, a(NN).
or
            a(0).
(2) ⟶      a(N) :- b, c(N).
            c(N) :- NN is N-1, a(NN).
```

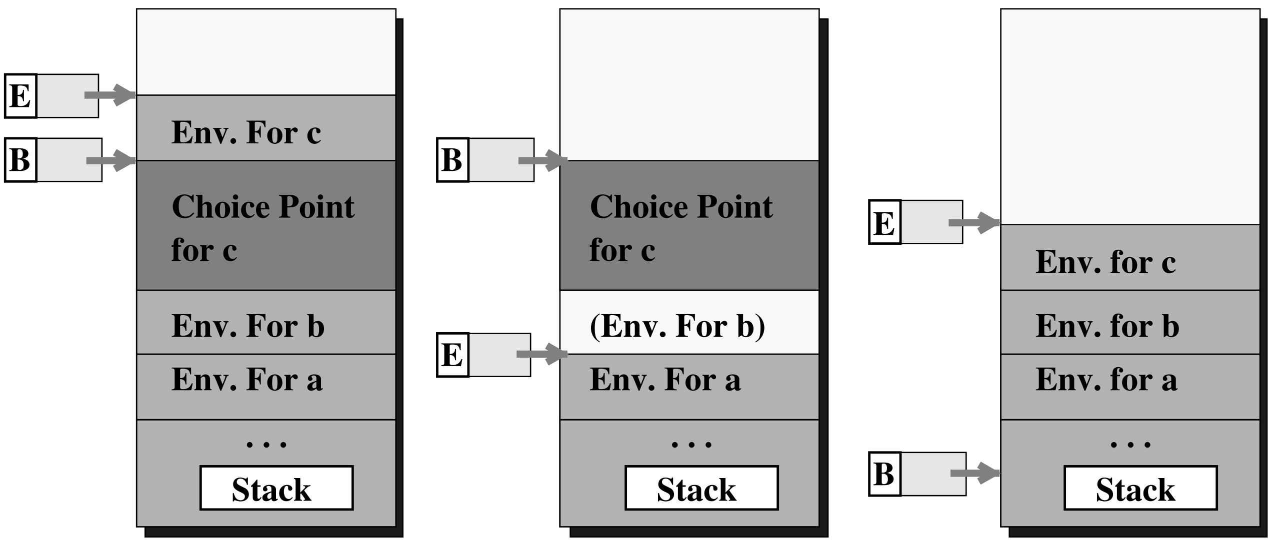


(1, no LCO) **(2, no LCO)** **(1, 2, LCO)**

"Environment Protection":

¤ Environments apparently deallocated can be preserved ("protected") by a Choice Point for reuse on backtracking:

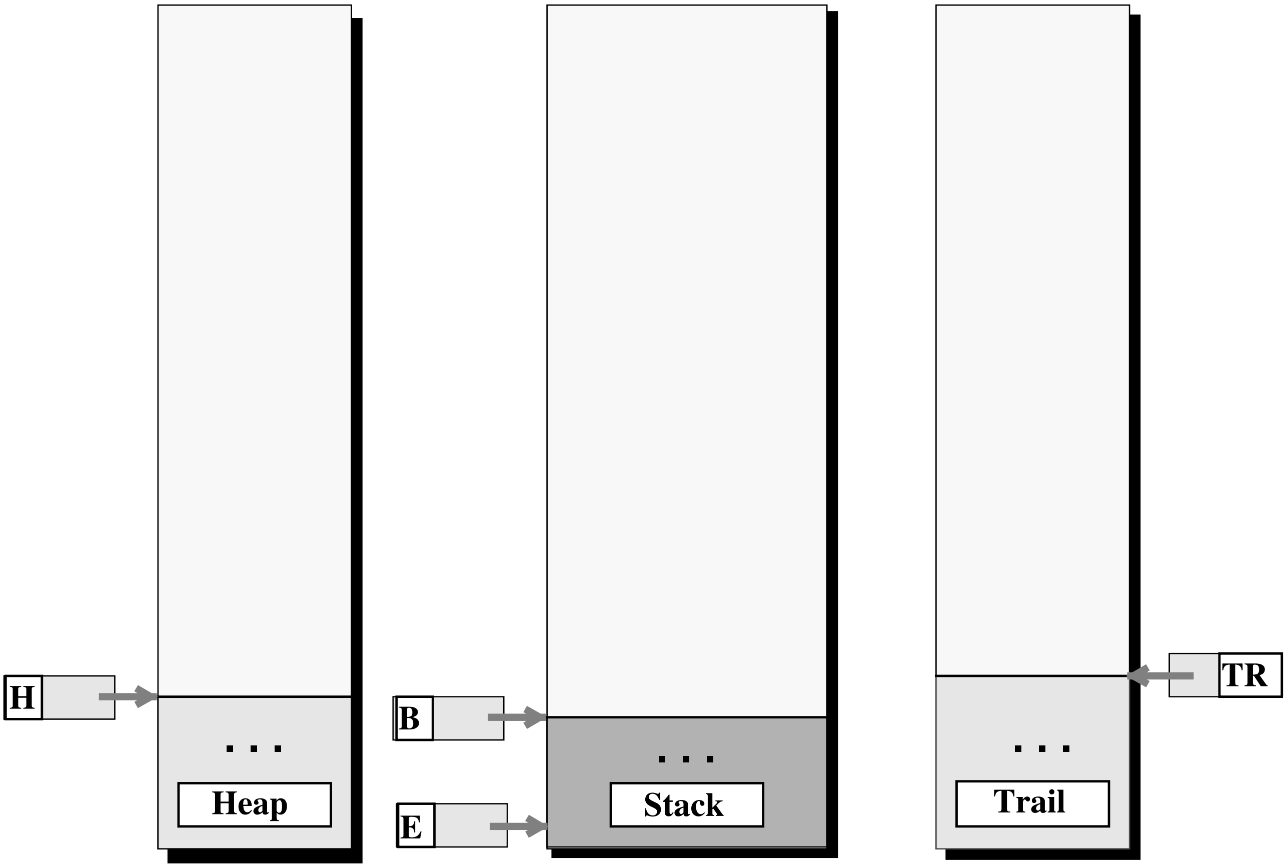
```
                 (2)
    a :- b, e.
    b :- c.
(1) c :- d, h.
(3) c :- d.
    d.
    e :- fail.
    h.
```



(1) (2) (3)

Backtracking: Control and storage recovery

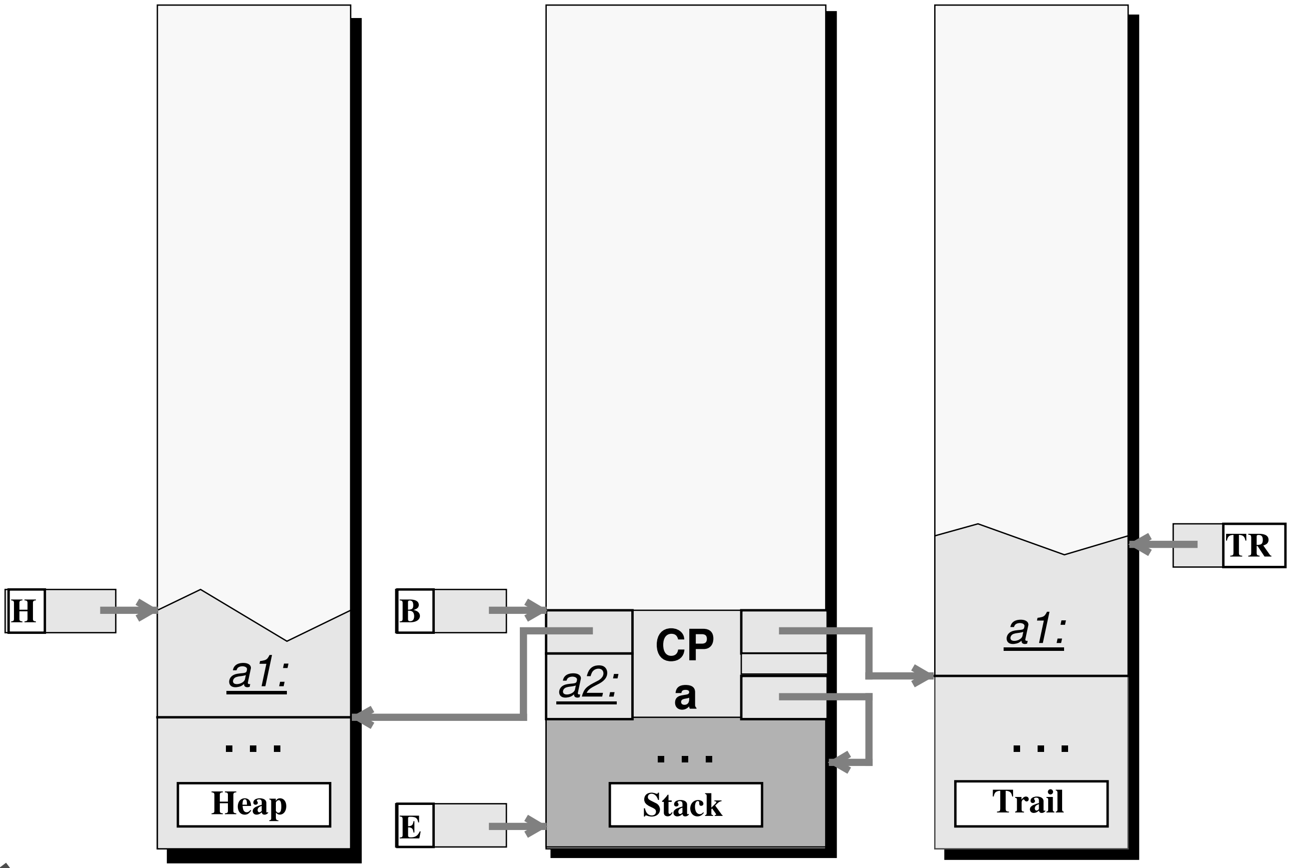
?:- a.



a1:	**a :- b, c, d.**	*b1:*	**b :- . . .**
a2:	**a :- b, c, d.**	*b2:*	**b :- . . .**
a3:	**a :- b, c, d.**	*b3:*	**b :- . . .**

Backtracking: Control and storage recovery

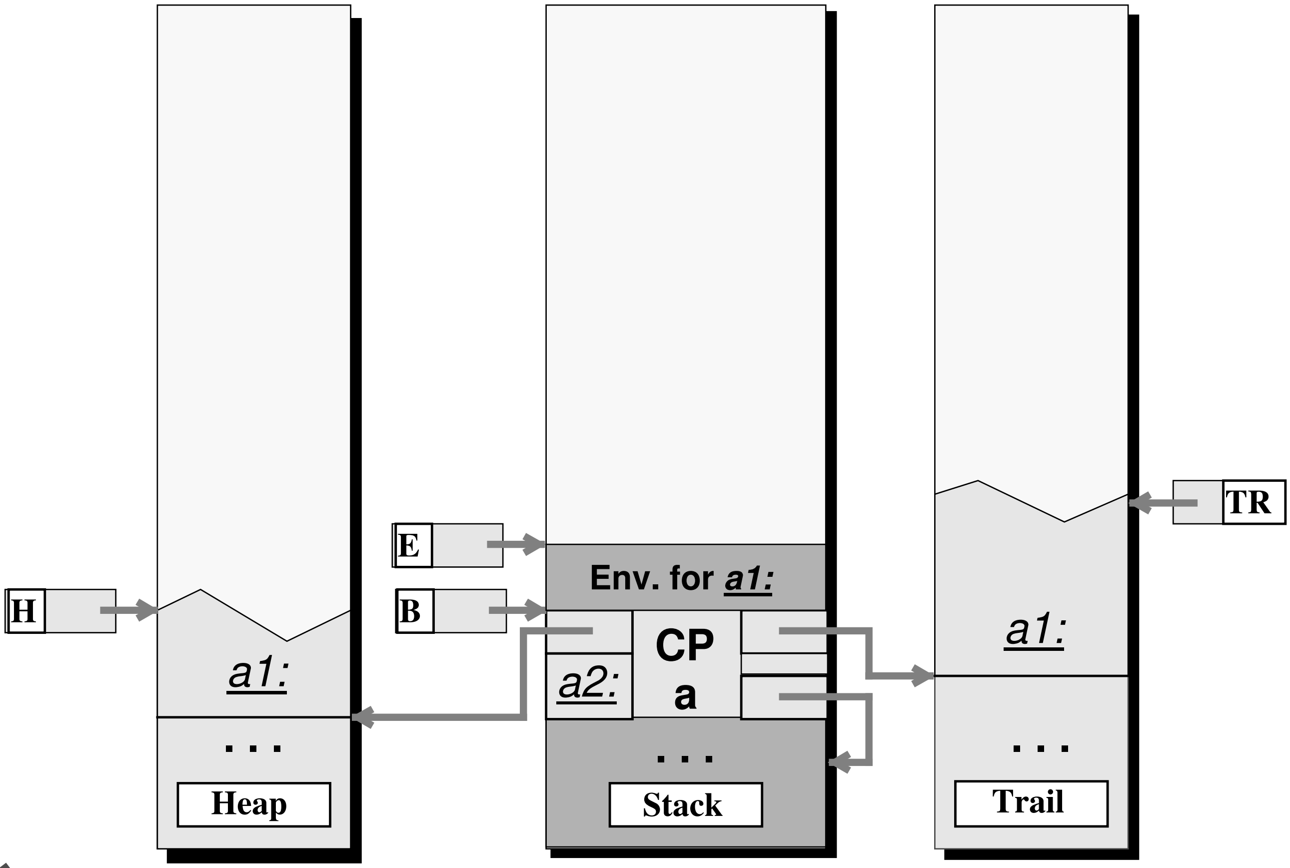
?:- a. **try**



a1:	**a :- b, c, d.**	*b1:*	**b :- . . .**
a2:	**a :- b, c, d.**	*b2:*	**b :- . . .**
a3:	**a :- b, c, d.**	*b3:*	**b :- . . .**

Backtracking: Control and storage recovery

?:- a. **allocate**

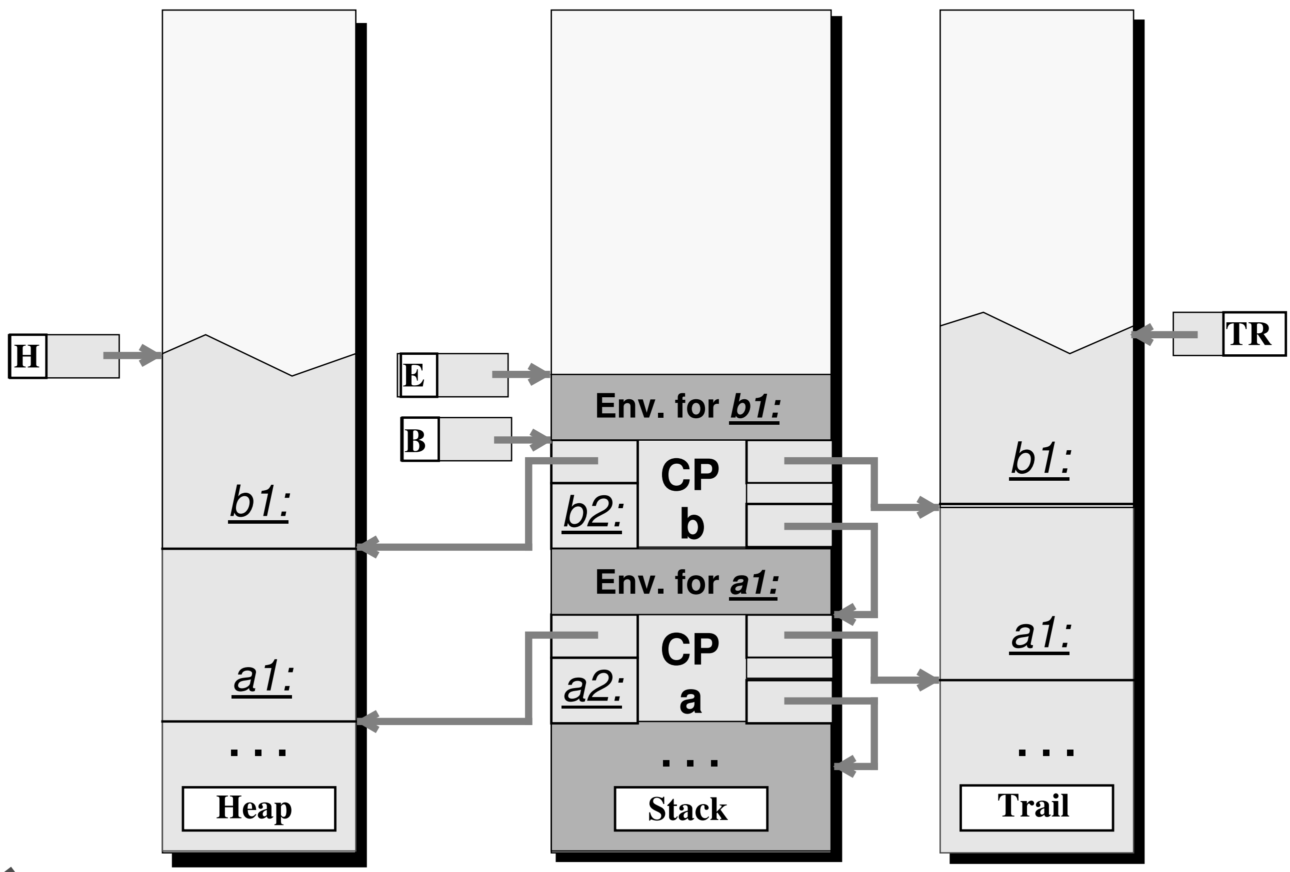


a1: →	**a :- b, c, d.**	*b1:*	**b :- . . .**
a2:	**a :- b, c, d.**	*b2:*	**b :- . . .**
a3:	**a :- b, c, d.**	*b3:*	**b :- . . .**

Backtracking: Control and storage recovery

?:- a.

try
allocate



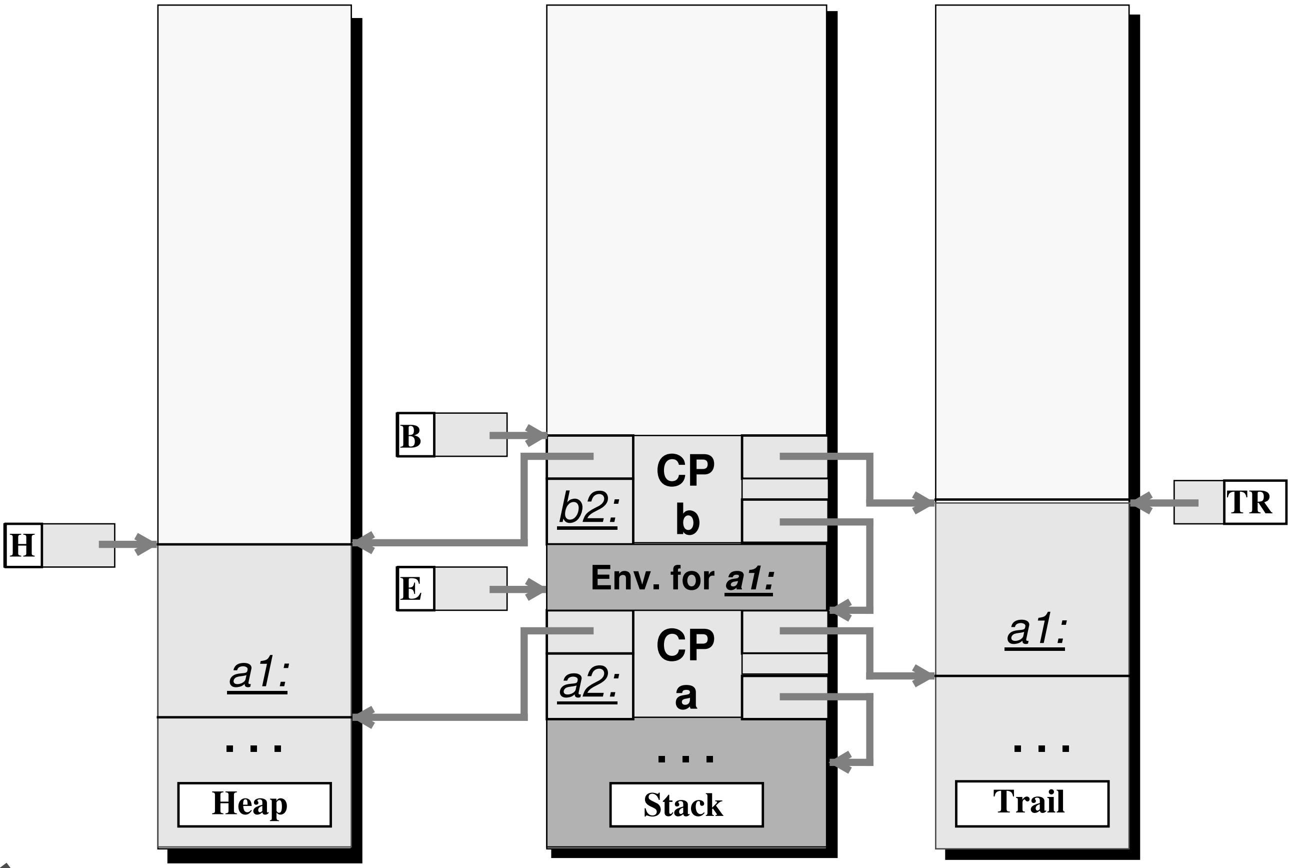
a1: → **a :- b, c, d.** → *b1:* → **b :- . . .**

a2: **a :- b, c, d.** *b2:* **b :- . . .**

a3: **a :- b, c, d.** *b3:* **b :- . . .**

Backtracking: Control and storage recovery

?:- a. **fail**

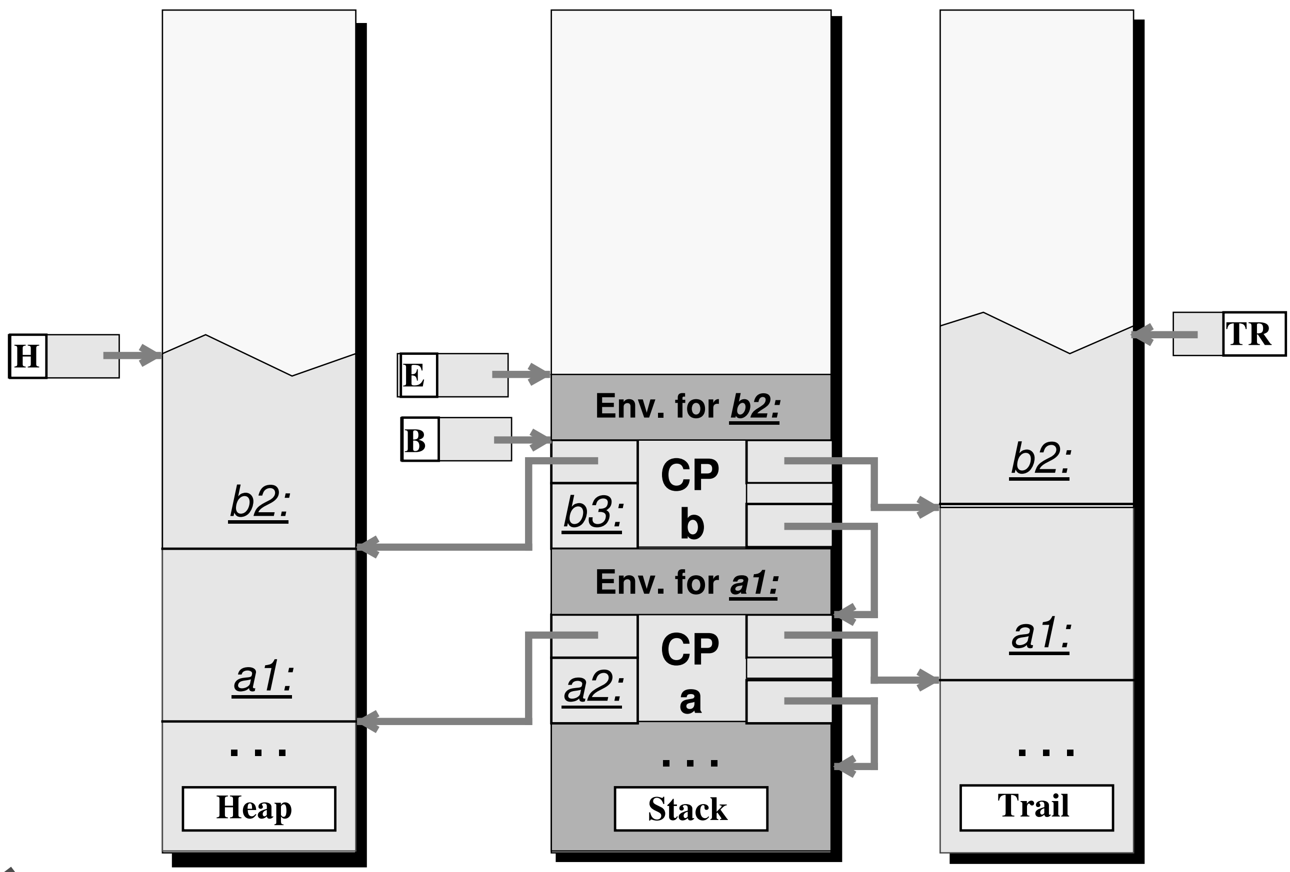


a1:	**a :- b, c, d.**	*b1:*	**b :- . . .** ***fail***
a2:	**a :- b, c, d.**	*b2:*	**b :- . . .**
a3:	**a :- b, c, d.**	*b3:*	**b :- . . .**

Backtracking: Control and storage recovery

?:- a.

try
allocate



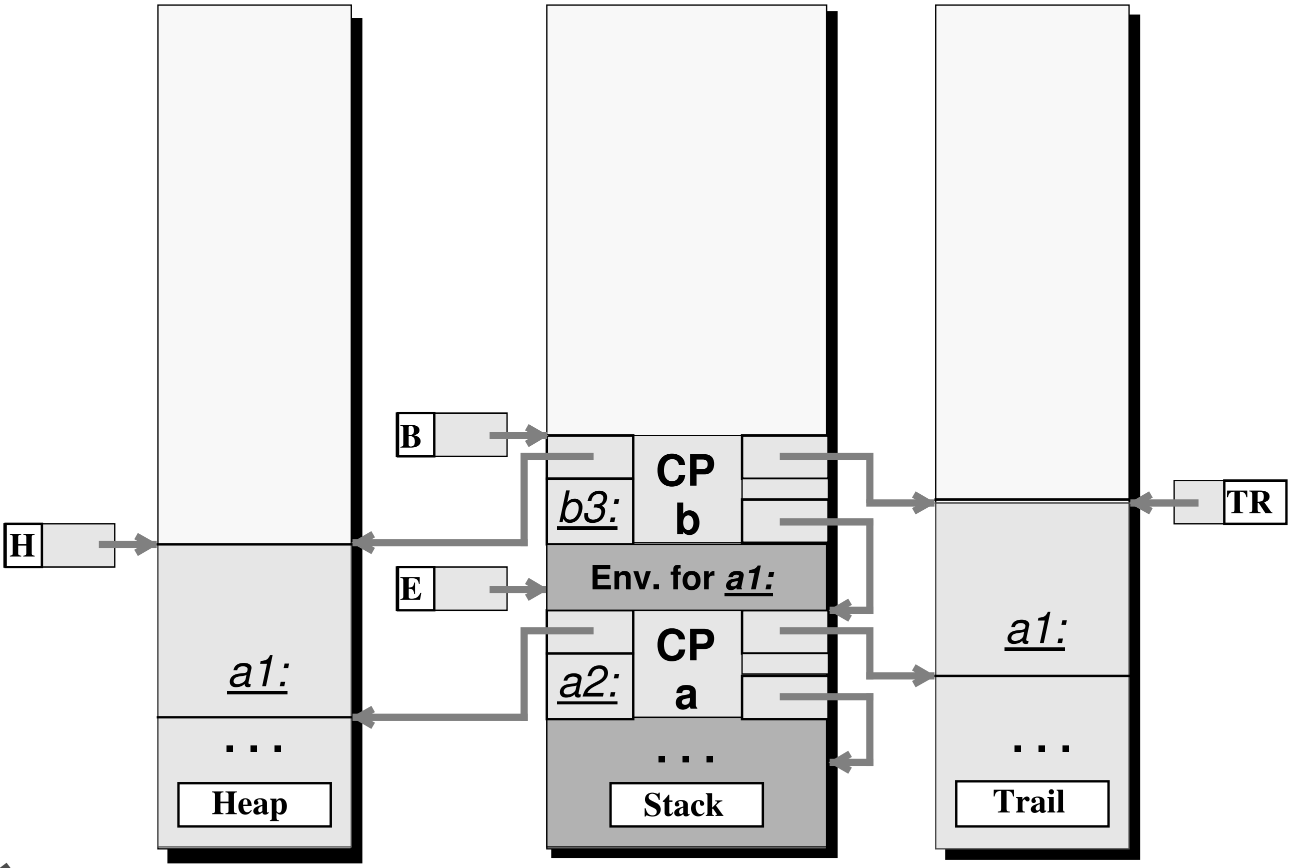
a1: → **a :- b, c, d.** → *b1:* → **b :- . . . *fail***

a2: **a :- b, c, d.** → *b2:* → **b :- . . .**

a3: **a :- b, c, d.** *b3:* **b :- . . .**

Backtracking: Control and storage recovery

?:- a. **fail**

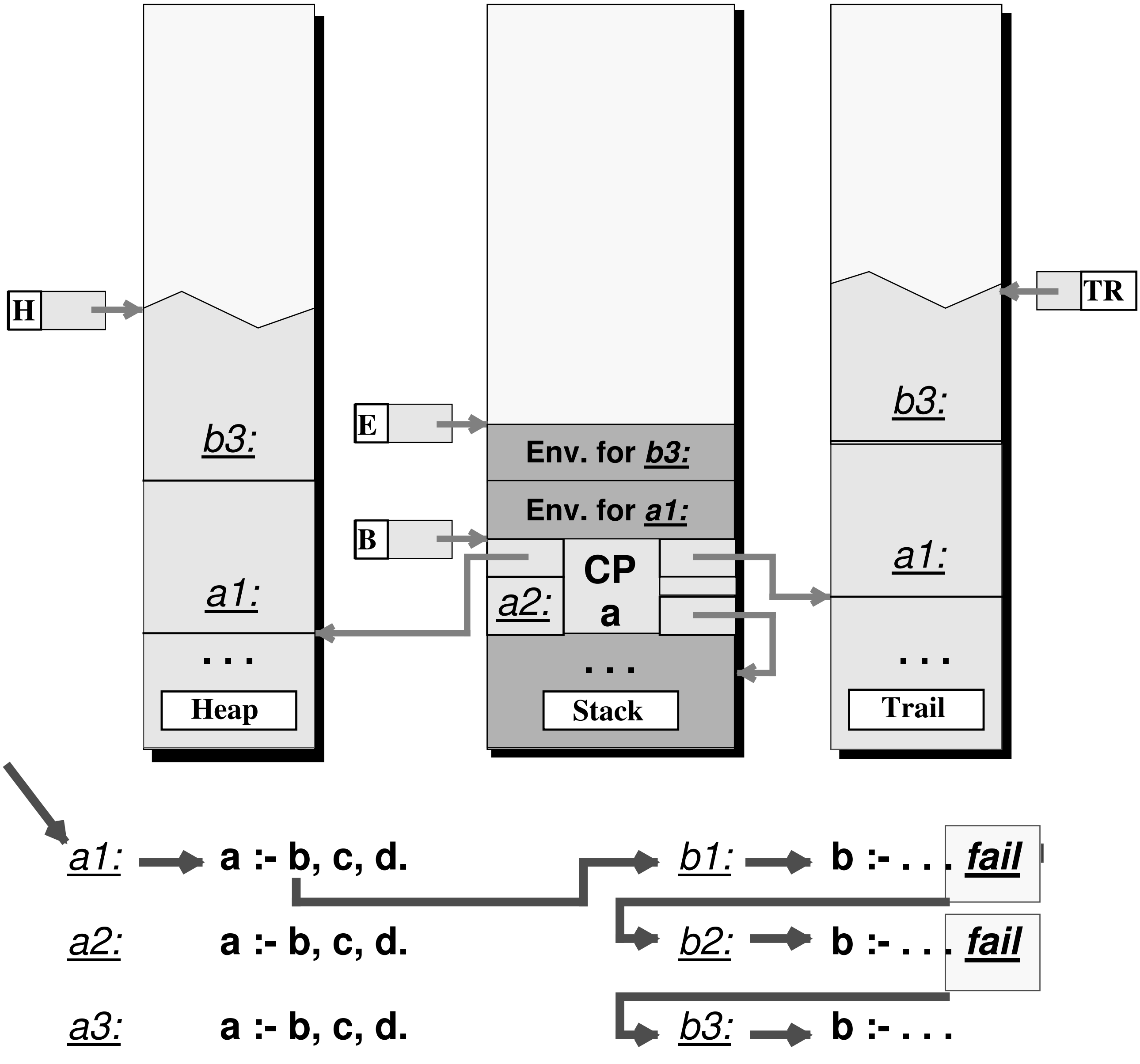


a1:	**a :- b, c, d.**	*b1:*	**b :- . . . *fail***
a2:	**a :- b, c, d.**	*b2:*	**b :- . . . *fail***
a3:	**a :- b, c, d.**	*b3:*	**b :- . . .**

Backtracking: Control and storage recovery

?:- a.

trust
allocate



The WAM Instruction Set (Simplified):

"**put**" instructions:

- *transfer arguments to argument regs.*

call / execute

- *procedure invocation*

allocate / deallocate

- *create / discard environments*

"**get**" instructions

- *get arguments from argument registers, unification ("customized"), failure*

"**unify**" instructions

- *full unification (read/write mode), failure*

proceed

- *return (success)*

try / retry / trust

- *create / update / discard choice points*

cut

switch (indexing) instructions**:**

```
switch_on_term Lv,Lc,Ll,Ls     (jump on tag)
switch_on_constant   N,table  (hashing)
switch_on_structure  N,table  (hashing)
```

WAM Code Example: append/3

append([],L,L).
append([H|T1],L2,[H|T2]):- append(T1,L2,T2).

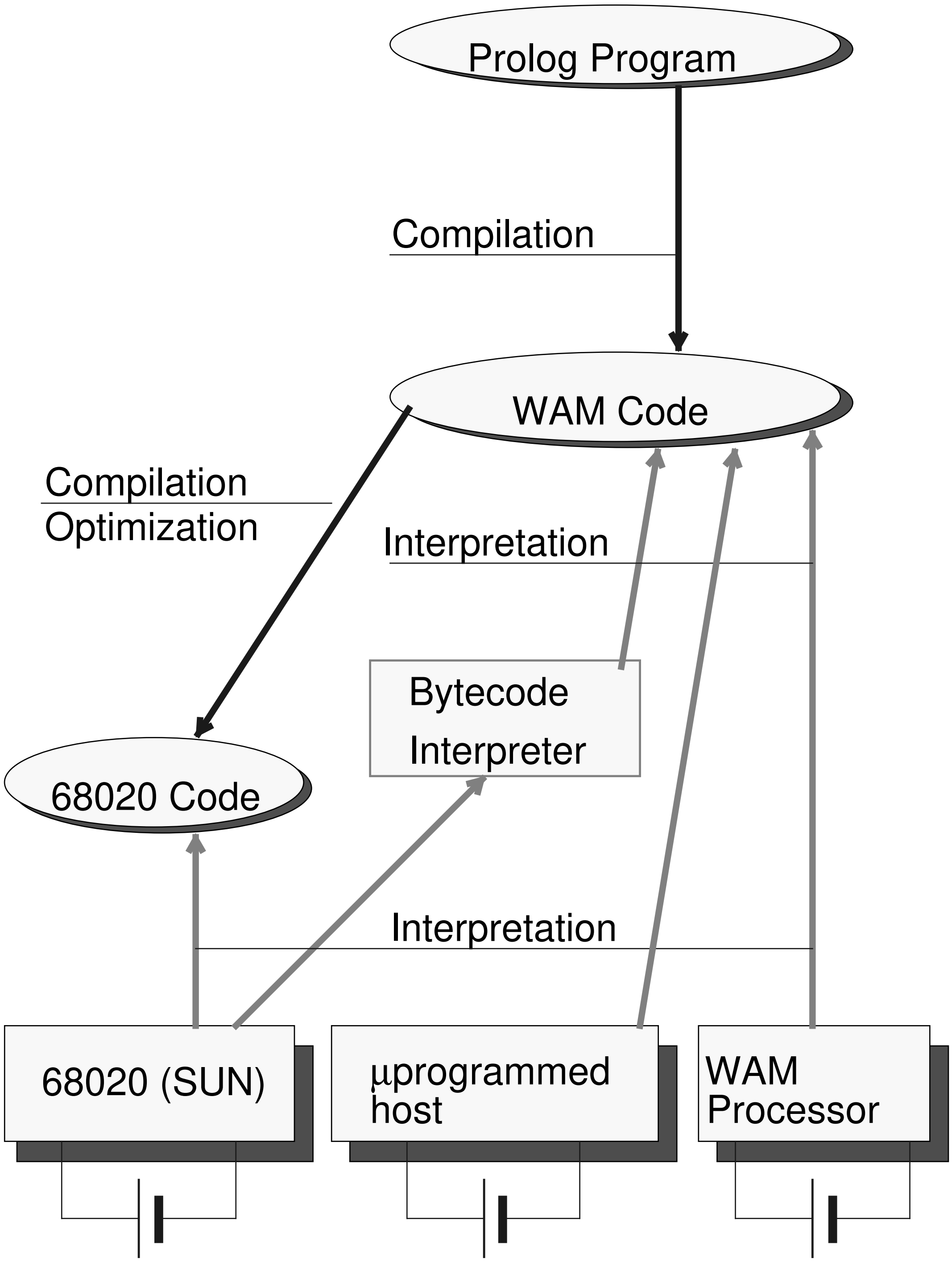
```
-------------------------------------------------------------------

          procedure          append/3

          switch_on_term     _951,_952,fail (const,list,struct) var
          try                3,_951
          trust              _952
_951:
          get_nil            X1                % [ ]
          get_value          X2,X3             % L,L
          proceed                              %
_952:
          get_list           X1                % [
          unify_variable     X4                %   H|
          unify_variable     X1                %     T1],L2,
          get_list           X3                % [
          unify_u_value      X4                %   H|
          unify_variable     X3                %      T2]
          execute            append/3          %

          end
```

WAM- Some Implementation Strategies:



WAM- Some Implementation Strategies:

Bytecode interpreters

- written in ‘C’ (e.g. SICStus, SB-Prolog, &-Prolog, PLM, Lcode, ...)
  + portability, small code size (≈ source)
  - speed (but it can be quite good with appropriate optimizations) (c.f. SICStus)
- written in assembler (e.g. Quintus Prolog)
  + speed (2x ‘C’ interpreter), small code size (≈ source)
  - needs to be rewritten for each architecture

Compilation to native code (e.g. BIM Prolog)

+ speed (in principle 2x assembler interpreter possible), extensive optimization possible
- code size, back-end rewrite for each architecture

µcoded WAM (e.g. Carlsson on LM’s, Gee et. al UCB ICLP87, ...):

+ small code size (≈ source), good performance (75% of PLM), original intent of the wam,
- writing µcode not easy, expensive host, µcoding more and more outdated...

WAM- Some Implementation Strategies: (contd.)

Compilation to ‘C’, a la KCL (e.g. Proteus Prolog)

+ good speed, extensive optimization possible, ‘C’ compiler optimization for free, portable
- modification to ‘C’ compiler needed for good performance, complex compiler, large code size (?)

Specialized Prolog machine (e.g. Xenologic, IPP, CHI-II, ECRC, ...)

+ high-performance potential, can be added as a co-processor to other machines
- first designs cost / reduced market, long design time, complexity of hardware debugging, difficulty in keeping up with technology generations, it is not clear yet what the ideal Prolog organization is...

Optimizations in the WAM:

Storage Efficiency:

- last call (“tail recursion”) optimization: deallocation of current environment *before* last call,
- selective allocation of choice points,
- space recovery on backtracking (auto GC),
- static/dynamic detection of unsafe vars.: put_unsafe_-value will "globalize" a dereferenced ptr. that lands in the current environment (because such a value may be destroyed by subsequent TRO),
- immediate reclamation of local storage (environment trimming): environments are "open-ended" and dynamically trimmed by overlaying callee’s environment

Execution Speed:

- efficient indexing (+ hashing on argument values),
- “customization” of unification,
- register allocation possible,
- fast backtracking,
- fast “cut,” etc.

WAM memory performance studies:

[Tick 88 - KAP] WAM Memory Referencing characteristics (data / instructions, CP / Env., caching approaches).

Conclusions:

- dereferencing chains are short.
- general unification is shallow.
- shallow backtracking major contributor to bandwidth requirement.
- small caches and local buffers quite effective.
- split-stack architecture efficient (2.5% extra references) method of simplifying architecture.
- ‘‘smart’’ cache gets largest savings by avoiding fetching the top of heap during structure writes. Second in savings is avoidance of copying-back of dead portions of the stack.
- Pascal benchmarks displayed lower traffic ratios for equal sized caches (for 1024 word caches):

  ° 2-word-lines: Pascal is 33% traffic of Prolog

  ° 4-word-lines: Pascal is 50% traffic of Prolog
- *best choice* Prolog local memories:

  ° low-cost (<16 words): choice point buffer

  ° medium-cost (32--128 words): stack buffer

  ° high-cost (>200 words): copyback caches

WAM memory performance studies:

[Touati et. al] (PLM -- UCB) Confirmation of some of Tick's conclusions and some new ones:

- savings in environment bandwidth can be attained by using a split-stack architecture and reusing top environments: for **Puzzle**, 52% of environment creations are "avoidable".
- large savings in choice point bandwidth can be attained by relatively simple compiler optimizations:
  for **N-Queens**, 25%--55% of choice point creations are "avoidable".
- cdr-coding is ineffective.

Touati and Despain - SLP87

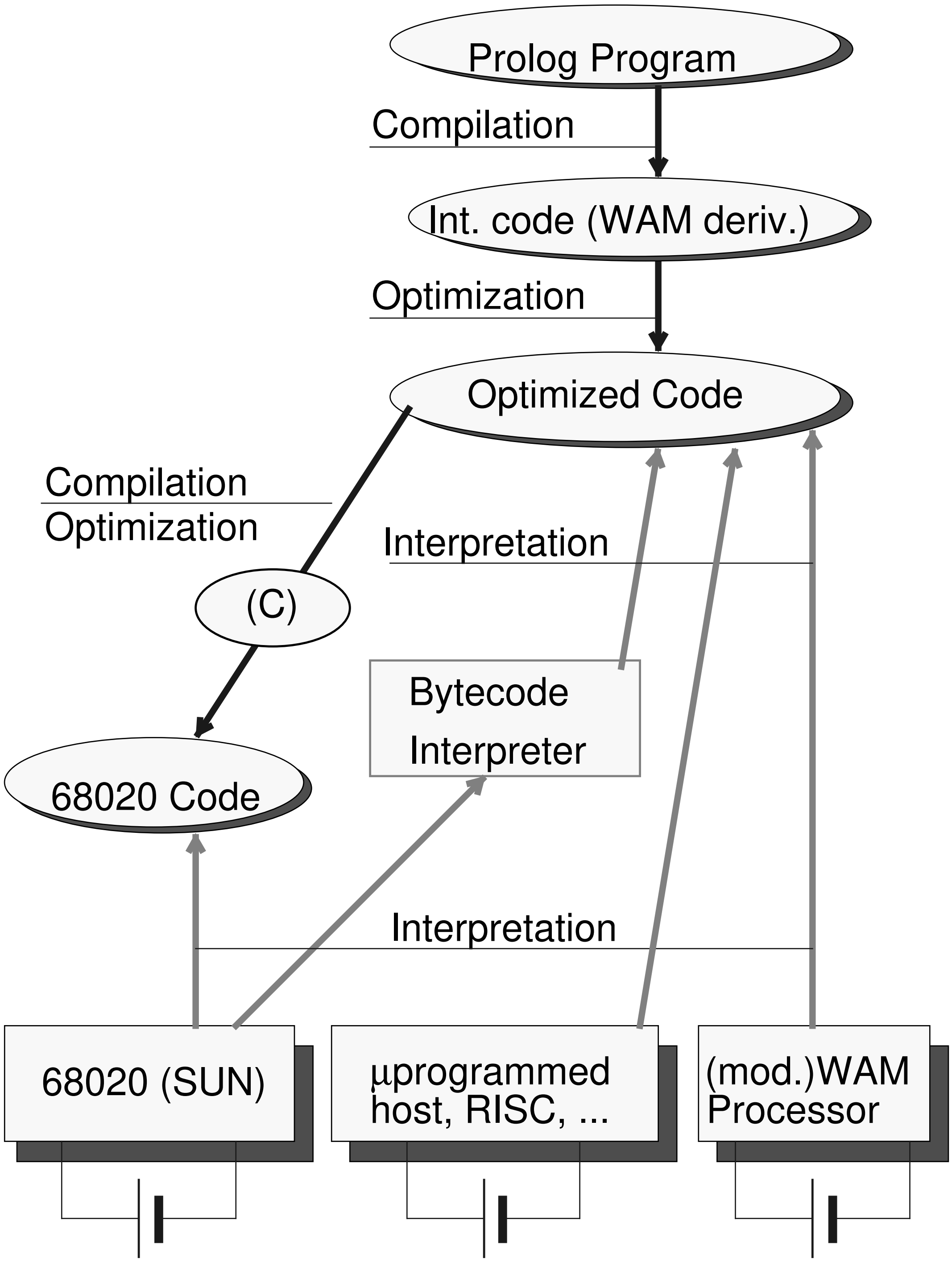
Other studies have obtained similar conclusions.

WAM Limitations Identified:

- arg. registers modified in head: shallow backtracking overhead,
- it is difficult to make use of mode information,
- indexing as described is simplistic: execution profile is sequence of jumps,
- abstract instruction set too high-level: restricts optimizations,
- environments and choice points allocated on same stack: reduces locality, increases complexity.
- *read* and *write* modes can cause complexity/inefficiency in emulator.
- architecture too complex, e.g., environment trimming, many pipeline breaks.

Not necessarily wrong, but due to the original execution target (µprogrammed CISC). Most newer proposals are *evolutions* of the WAM.

Mod-WAM Implementation Strategies:



Some Special-purpose Sequential Prolog Machines

Machine	Group	Language	Comments
PSI-I	ICOT/Mitsub.	Prolog(ESP)	microcoded (mc) interpreter
PSI-II	ICOT/Mitsub.	Prolog(ESP)	mc super-CISC WAM
CHI-I	NEC	Prolog	mc WAM co-proc.
CHI-II	NEC	Prolog	mc super-CISC WAM co-proc.
PLM	UCB	Prolog	mc WAM co-proc.
X1	XenoLogic	Prolog	mc WAM co-proc.
IPP	Hitachi	Prolog	supermini-based mc/mod WAM
IP704	Toshiba	Prolog	mc WAM co-proc
Pegasus	Mitsubishi	Prolog	tagged-RISC
MAIA	CNET/CGE	Prolog	mc Lisp machine
KCM	ECRC	Prolog	mc mod WAM
Low-RISC	Indiana U.	Prolog	mod WAM / native RISC
PLUM	UCB	Prolog	mod WAM
ICM4	ECRC	Prolog	RISC

Some Special-purpose Parallel Prolog Machines

PIM-D	Oki	Prolog	AND/OR dataflow
PIM-R	Hitachi	Prolog	AND/OR reduction
Kabuwake	Fujitsu	Prolog	OR-parallel
Aquarius-2	UCB	Prolog/...	PPPs on a crossbar (proposed)
DDM	Bristol/Sics	Prolog/...	Shared virtual address space

Some Interesting Host Implementations

SUNS etc.	Quintus	Prolog	Q Prolog - WAM, Industry standard
SUNS etc.	BIM	Prolog	native code, WAM+opt, high-perf
SUNS etc.	SUNY	Prolog	SB-Prolog, WAM, public domain
SUNS etc.	UCB PLM	Prolog	WAM, public domain
SUNS etc.	SICStus	Prolog	Portable mod WAM, good perf.
SPUR	UCB	Prolog	native-coded WAM on tag-RISC
VAX-8600	UCB	Prolog	mc WAM on general purpose
Symmetry	Gigalips	Prolog	OR-parallel WAM emulator
Symmetry	MCC/UT	Prolog	Ind. AND-parallel RAP-WAM em.
Transp.	Parsytec	Prolog	Ind. AND-parallel RAP-WAM

Relative Speeds
(absolute speed is of course cycle dependent)

Examples (circa 1989):

1.-	BIM-Prolog	200	Klips
	Sicstus-Prolog (native)	200	Klips
2.-	Quintus-Prolog	100	Klips
	Sicstus-Prolog	80	Klips
	SB-Prolog	30	Klips
3.-	Hitachi IPP	1000	Klips
	ECRC ICM-3	530	Klips
	CHI-II	500	Klips
	Xenologic X1	300	Klips
	ICOT PSI-II	250	Klips

Global Analysis of Logic Programs:

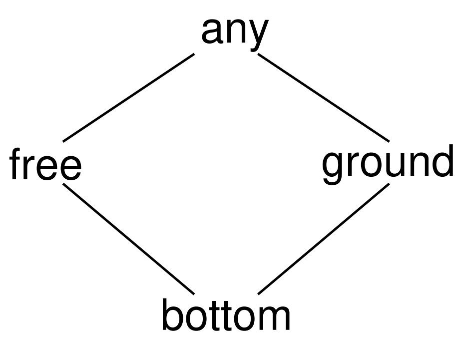
```
p(X,Y) :- q(X,Y).
q(W,W).
```

- Could be done by collecting all possible substitutions at each point in the program: but, given that there are term constructors in the language the set can be infinite $\rightarrow$ non-terminating computation.
- Abstract interpretation: use "abstract substitutions" instead of actual substitutions
- Abstract substitution: an element of an abstract domain is associated with each variable. (Other approaches are also possible)
- Elements of the abstract domain are finite representations of possibly infinite sets of actual substitutions/terms
- The abstract domain is generally a partial order or cpo of finite height (termination), "$\leq$"
- Abstraction function α: set of concrete substitutions $\rightarrow$ abstract substitution
- Concretization function γ: abstract substitution $\rightarrow$ set of concrete substitutions
- For each operation u (e.g. unification) of the language there is a corresponding abstract operation u'
- Soundness requires that for all x in the abstract domain $u(x) \subseteq \gamma(u'(\alpha(x)))$

Simple Example

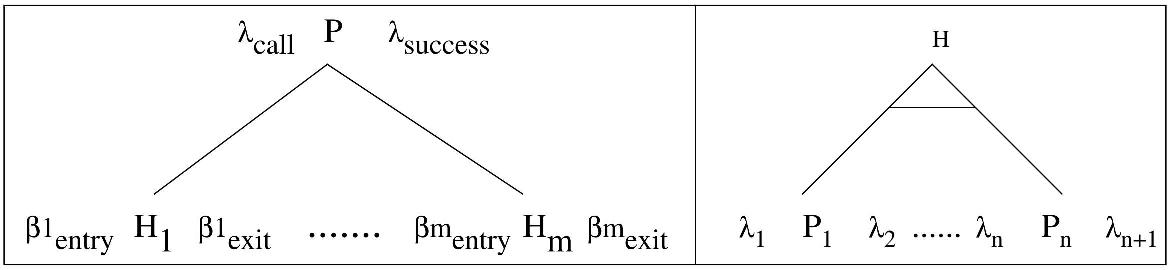
- A simple abstract domain for PROLOG
  = *{free, ground, any, bottom}*
- all ground terms → ground
- all terms → any
- all unbound variables → free
- bottom = ∅, i.e. failure

Partial order: corresponding to set inclusion in the actual domain:



Abstract interpretation procedure:

- The analysis starts with a set of clauses and one or more "query forms" (not strictly required).
- The goal of the abstract interpreter is to compute in abstract form the set of substitutions which can occur at each point in the program, during the execution of all queries that are concretizations of the query forms.
- Control: one solution is to build an abstract AND/OR tree (top-down):



- The key issues are related to abstract unification:

  ° computing entry subst. from call subst.

  ° computing success subst. from exit subst.

  ° success substitutions from alternative branches are then combined (LUB).

- Recursion: consider a recursive predicate p such that there are two identical or-nodes for p, one an ancestor of the other, and with identical call substitutions $\rightarrow$ infinite loop.
- Fixpoint calculation required (several alternatives).

Abstract Interpretation: Issues

- Sound mathematical setting [Cousot and Cousot 77]
- Extended to flow analysis of logic programs [Bruynooghe, Jones and Sondergaard, Mellish], proved termination properties given certain constraints imposed on the abstract domain and operations
- Specific algorithms and applications [Debray and Warren "abstract compilation", Mannila and Ukkonen, Mellish jlp2, Sondergaard iclp88, Bruynooghe GC slp87, Sato and Tamaki, Waern, Warren and Hermenegildo, Muthukumar and Hermenegildo...]
- Difficult issues: dealing with dynamic predicates [Debray slp87]
- Abstract interpretation has been shown to be a practical compilation tool [Warren / Hermenegildo / Debray iclp88], also description of tradeoffs in efficient implementation
- Important application: support for smart computation rules - "optimization by not doing the work, rather than by doing it faster" Freeze, NU-Prolog, ... See Andorra, later.
- Important issue: correct, precise, and efficient tracking of variable aliasing [Debray, Bruynooghe, Jacobs and Langen, Muthukumar and Hermenegildo NACLP89, ...]
- Important issue: sharing + freeness [Muthukumar and Hermenegildo ICLP91, ...]
- See [Carlsson, Debray, Marien et al., Taylor et al.] in ICLP '89, ICLP'90, NACLP90.

Issues in High Performance Prolog Implementation:

- Instruction Set Design
  - ° WAM-based engines
  - ° RISC/CISC designs from WAM
- Compiler optimizations, global analysis (abs. interp.)
- Storage Model and Memory Performance
  - ° memory bandwidth requirements
  - ° local memory behavior and characteristics
  - ° stack-, tree-, heap-based memory management
  - ° locking requirements
- Efficiency of Fundamental Operations:
  unification, dereferencing, binding, backtracking, cut
- Efficiency of Parallel Management
  - ° spawning a process/switching a task
  - ° scheduling: suspension/resumption
  - ° load balancing
- Available Parallelism
  - ° tradeoff between availability and programmability.
  - ° issues in automatic parallelization
  - ° AND/OR, extension to dep. and-parallelism