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  Battani-Meloni Interpreter Structure-sharing
       Marseille Prolog in Fortran
       ↓
1977  DEC-10 Prolog Compiler to Structure-sharing,
       Edinburgh native code multiple stacks:
       ↓→ Icot Machine (PSI)
       ↓
       Portable Prolog Compiler [Bowen et. al] ...
       ↓
1983  "Old Engine" compiler to structure copying,
       SRI abstract machine goal stacking
       ↓ code + emulator
       ↓
1983/4 "New Engine" compiler to structure copying,
       SRI (WAM) abstract machine environment stacking,
       ↓ code + emulator 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 ef-
ficiency.

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

ARGUMENT AND MACHINE REGS.

AX1
AX2

|

AXn

P
CP

PROGRAM AND DATA AREAS

Code

PDL

STACK

TRAIL

HEAP

CONTENTS OF DATA AREAS

WAM Instructions

CE (Prev. En.)
CP (parent CP)
Y1 Register

Yn Register

Environments

B’ (Prv. Ch.P.)
BP (Next. Alt.)

E
H
TR
CP
N
AX1

Choice Points

Variable Addresses

Lists and Structures

Globalized Variables

Environments

WAM Instructions

CE (Prev. En.)
CP (parent CP)
Y1 Register

Yn Register

Environments

B’ (Prv. Ch.P.)
BP (Next. Alt.)

E
H
TR
CP
N
AX1

Choice Points

Variable Addresses

Lists and Structures

Globalized Variables
### Data Types:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
<td>value</td>
</tr>
<tr>
<td>ref</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>const</td>
<td>a</td>
</tr>
<tr>
<td>const</td>
<td>b</td>
</tr>
<tr>
<td>const</td>
<td>c</td>
</tr>
</tbody>
</table>

1. **Reference:** represents variables.

   - **Unbound var**
   - **Bound var**

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

   - "a" **const** a

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

   ```
   struct
   "foo(a, b, c)"
   ```

4. **List:** special case of structure.

   ```
   list
   "[a, b, c]"
   ". (a, . ( b, . ( c, [ ] )))"
   ```
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).
```

![Diagram](image)
**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).
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_list</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>unify_variable</td>
<td>X4</td>
<td>%</td>
</tr>
<tr>
<td>unify_variable</td>
<td>A1</td>
<td>%</td>
</tr>
<tr>
<td>get_list</td>
<td>A3</td>
<td>%</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>%</td>
</tr>
</tbody>
</table>
```

**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).
```

Stack diagrams:
- **(1, no LCO)**: Environment for `a`
  - Environment for `a`
  - Environment for `a`
  - Environment for `a`
  - Stack

- **(2, no LCO)**: Environment for `c`
  - Environment for `c`
  - Environment for `c`
  - Environment for `c`
  - Environment for `c`
  - Stack

- **(1, 2, LCO)**: Stack

School of Computer Science (FI) Technical University of Madrid (UPM)
"Environment Protection":

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

\[
\begin{align*}
(1) & \quad \rightarrow \quad c \leftarrow d, h. \\
(2) & \quad \rightarrow \quad a \leftarrow b, e. \\
(3) & \quad \rightarrow \quad c \leftarrow d. \\
& \quad \quad \downarrow \quad \downarrow \quad \downarrow \\
& \quad b \leftarrow c. \\
& \quad c \leftarrow d. \\
& \quad d. \\
& \quad e \leftarrow \text{fail.} \\
& \quad h.
\end{align*}
\]
Backtracking: Control and storage recovery

?- a.
Backtracking: Control and storage recovery

?- a.

try

a1: a :- b, c, d.

a2: a :- b, c, d.

a3: a :- b, c, d.

b1: b :- . . .

b2: b :- . . .

b3: b :- . . .
**Backtracking**: Control and storage recovery

?- a.

allocate

a1: a :- b, c, d.

a2: a :- b, c, d.

a3: a :- b, c, d.

b1: b :- .

b2: b :- .

b3: b :- .

allocate

CP

E

B

H

Env. for a1:

Trail

Allocate Env. for a1:

Trail

Heap

Stack

Trail

allocate

Env. for a1:

Trail

Heap

Stack

Trail

allocate

Env. for a1:

Trail
Backtracking: Control and storage recovery

?- a.

\[ a1: \quad a :- b, c, d. \]
\[ b1: \]
\[ b2: \quad b :- . . . \]
\[ a1: \]
\[ a2: \quad a :- b, c, d. \]
\[ b1: \]
\[ b2: \quad b :- . . . \]
\[ a1: \]
\[ a3: \quad a :- b, c, d. \]
\[ b1: \]
\[ b2: \quad b :- . . . \]
\[ a1: \]

CP

a

Env. for \( b1: \)

CP

b

Env. for \( a1: \)

CP

a

Stack

Heap

Env. for \( a1: \)

Env. for \( b1: \)

allocate

try

School of Computer Science (FI) Technical University of Madrid (UPM)
Backtracking: Control and storage recovery

?- a.
fail

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

CP

Env. for a1:

TR

H

Heap

Stack

Trail

School of Computer Science (FI) Technical University of Madrid (UPM)
Backtracking: Control and storage recovery

?- a.

\[ a1: a :- b, c, d. \]
\[ a2: a :- b, c, d. \]
\[ a3: a :- b, c, d. \]

\[ b1: \text{fail} \]
\[ b2: \]
\[ b3: \]

try allocate

\[ H \rightarrow \text{Heap} \]
\[ E \rightarrow \text{Env. for } b2: \]
\[ CP \rightarrow \text{b} \]
\[ TR \rightarrow \text{Trail} \]
Backtracking: Control and storage recovery

?:- a.

fail

Env. for a1:

CP

b

Stack

H

E

CP

a

Heap

a1:

... 

Heap

a2:

... 

Stack

a3:

... 

Heap

a1:

a :- b, c, d.

b1:

b :- ...

fail

b3:

b :- ...

fail

b2:

b :- ...

fail

b3:

b :- ...

TR

Env. for a1:
**Backtracking:** Control and storage recovery

?- a.

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

Diagram:

- **Stack**
- **Heap**
- **Trail**
- **Env. for a1:**
- **Env. for b3:**
- **CP**
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,
going_list X3 % [
unify_u_value X4 % H|
unify_variable X3 % T2]
execute append/3 %

end
WAM - Some Implementation Strategies:

- Prolog Program
- 68020 Code
- WAM Code
- Bytecode Interpreter
- WAM Processor
- µprogrammed host
- 68020 (SUN)
- Compilation
- Compilation Optimization
- Interpretation
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

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

### Some Special-purpose Parallel Prolog Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Group</th>
<th>Language</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIM-D</td>
<td>Oki</td>
<td>Prolog</td>
<td>AND/OR dataflow</td>
</tr>
<tr>
<td>PIM-R</td>
<td>Hitachi</td>
<td>Prolog</td>
<td>AND/OR reduction</td>
</tr>
<tr>
<td>Kabuwake</td>
<td>Fujitsu</td>
<td>Prolog</td>
<td>OR-parallel</td>
</tr>
<tr>
<td>Aquarius-2</td>
<td>UCB</td>
<td>Prolog/...</td>
<td>PPPs on a crossbar (proposed)</td>
</tr>
<tr>
<td>DDM</td>
<td>Bristol/Sics</td>
<td>Prolog</td>
<td>Shared virtual address space</td>
</tr>
</tbody>
</table>

### Some Interesting Host Implementations

<table>
<thead>
<tr>
<th>SUNS etc.</th>
<th>Quintus</th>
<th>Prolog</th>
<th>Q Prolog - WAM, Industry standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUNS etc.</td>
<td>BIM</td>
<td>Prolog</td>
<td>native code, WAM+opt, high-perf</td>
</tr>
<tr>
<td>SUNS etc.</td>
<td>SUNY</td>
<td>Prolog</td>
<td>SB-Prolog, WAM, public domain</td>
</tr>
<tr>
<td>SUNS etc.</td>
<td>UCBM PLM</td>
<td>Prolog</td>
<td>WAM, public domain</td>
</tr>
<tr>
<td>SUNS etc.</td>
<td>SICStus</td>
<td>Prolog</td>
<td>Portable mod WAM, good perf.</td>
</tr>
<tr>
<td>SPUR</td>
<td>UCB</td>
<td>Prolog</td>
<td>native-coded WAM on tag-RISC</td>
</tr>
<tr>
<td>VAX-8600</td>
<td>UCB</td>
<td>Prolog</td>
<td>mc WAM on general purpose</td>
</tr>
<tr>
<td>Symmetry</td>
<td>Gigalips</td>
<td>Prolog</td>
<td>OR-parallel WAM emulator</td>
</tr>
<tr>
<td>Symmetry</td>
<td>MCC/UT</td>
<td>Prolog</td>
<td>Ind. AND-parallel RAP-WAM em.</td>
</tr>
<tr>
<td>Transp.</td>
<td>Parsytec</td>
<td>Prolog</td>
<td>Ind. AND-parallel RAP-WAM</td>
</tr>
</tbody>
</table>
Relative Speeds
(absolute speed is of course cycle dependent)

Examples (circa 1989):

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BIM-Prolog</td>
<td>200 Klips</td>
</tr>
<tr>
<td></td>
<td>Sicstus-Prolog (native)</td>
<td>200 Klips</td>
</tr>
<tr>
<td>2.</td>
<td>Quintus-Prolog</td>
<td>100 Klips</td>
</tr>
<tr>
<td></td>
<td>Sicstus-Prolog</td>
<td>80 Klips</td>
</tr>
<tr>
<td></td>
<td>SB-Prolog</td>
<td>30 Klips</td>
</tr>
<tr>
<td>3.</td>
<td>Hitachi IPP</td>
<td>1000 Klips</td>
</tr>
<tr>
<td></td>
<td>ECRC ICM-3</td>
<td>530 Klips</td>
</tr>
<tr>
<td></td>
<td>CHI-II</td>
<td>500 Klips</td>
</tr>
<tr>
<td></td>
<td>Xenologic X1</td>
<td>300 Klips</td>
</tr>
<tr>
<td></td>
<td>ICOT PSI-II</td>
<td>250 Klips</td>
</tr>
</tbody>
</table>
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 \(\alpha\): set of concrete substitutions \(\rightarrow\) abstract substitution
- Concretization function \(\gamma\): 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
  \[ \{ \text{free}, \text{ground}, \text{any}, \text{bottom} \} \]
- all ground terms \(\rightarrow\) ground
- all terms \(\rightarrow\) any
- all unbound variables \(\rightarrow\) free
- \(\text{bottom} = \emptyset\), i.e. failure

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

```
    any
     / \   /
  free  ground
     \   /
    \ bottom
```
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):

```
\begin{array}{c}
\lambda_{\text{call}} & P & \lambda_{\text{success}} \\
\beta_{1}^{\text{entry}} & H_{1} & \beta_{1}^{\text{exit}} & \cdots & \beta_{m}^{\text{entry}} & H_{m} & \beta_{m}^{\text{exit}} \\
\end{array}
```

- 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