Implementação de Linguagens

Warren's Abstract Machine

Ricardo Rocha
DCC-FCUP, Universidade do Porto
ricroc @ dcc.fc.up.pt

Introduction

- Warren's Abstract Machine (WAM) was specified in 1983 by David H. D. Warren.
- This course consists of a *gradual* reconstruction of the WAM through several intermediate abstract machine designs:
  - Language $\mathcal{L}_0$: Unification

    One predicate defined by one fact.
  - Language $\mathcal{L}_1$: Unification

    Several predicates each defined by one fact.
  - Language $\mathcal{L}_2$: Flat Resolution (Prolog without backtracking)

    Several predicates each defined by one rule.
  - Language $\mathcal{L}_3$: Pure Prolog (Prolog with backtracking)

    Several predicates each defined by several rules.
  - Optimizations

First Order Terms

- A *variable* denoted by a capitalized identifier:
  - *X, X1, Y, Constant, ...*
- A *constant* denoted by an identifier starting with a lower-case letter:
  - *a, b, variable, cONSTANT, ...*
- A *structure* of the form $f(t_1,...,t_n)$ where *f* is a symbol called a *functor* (denoted like a constant) and the t_i's are first-order terms:
  - *f*(*X*), *p(Z, h(Z,W), f(W)), ...*
  - *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 *p*
  - a *query* term *?- q*

  where *p/q* are *non-variable* first-order terms (the scope of variables is limited to a program/query term).

Semantics

- Computation of the *Most General Unifier* (MGU) of program *p* and query *?- q*:
  - Either execution fails if *p* and *q* do not unify (in $\mathcal{L}_0$ failure aborts all further work).
  - Or it succeeds with a binding of the variables in *q* obtained by unifying it with *p*.

Abstract Machine $\mathcal{M}_0$: Heap Data Cells

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

- *variable cell* <REF,*k*> where *k* is a store address (an index into HEAP).

  An *unbound variable* at address *k* is <REF,*k*>.

- *structure cell* <STR,*k*> where *k* is the address of a functor cell.

  A structure $f(t_1,...,t_n)$ takes $n+2$ heap cells. The first cell of $f(t_1,...,t_n)$ is <STR,*k*>, where *k* is the address of a (possibly non-contiguous) functor cell containing *f/n*.

- *functor cell f/n (untagged)* where *f* is a functor symbol and *n* is its arity.

  A functor cell is *always* immediately followed by *n* contiguous cells, *i.e.*, if HEAP[*k*]=*f/n* then HEAP[*k+1*] refers to t_1, ..., and HEAP[*k+n*] refers to t_n.

Abstract Machine $\mathcal{M}_0$: Representation of Terms

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

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

Abstract Machine $\mathcal{M}_0$: Variable Registers

Variables *X1*, *X2*, ... are used to store temporarily heap data cells as terms are built.

- They are allocated to a term, one for each subterms.
- Variable registers are allocated according to least available index.
- Register *X1* is always allocated to the outermost term.
- The same register is allocated to all the occurrences of a given variable.

- The variable registers allocated to term *p(Z, h(Z,W), f(W))* are:
  - *X1 = p(X2,X3,X4)*
  - *X2 = Z*
  - *X3 = h(X2,X5)*
  - *X4 = f(X5)*
  - *X5 = W*

Abstract Machine $\mathcal{M}_0$: Flattened Form

- A term is equivalent to a conjunctive set of register assignments of the form:
  - $Xi = Var$
  - $Xi = f(Xi_1,\ldots,Xi_n)$
- The register assignments of the form $Xi = Var$ are meaningless.
- The *flattened form* of a query term is the *ordered* sequence of register assignments of the form $Xi = f(Xi_1,\ldots,Xi_n)$ so that a variable register is assigned *before* it is used as an argument in a subterm.

- The flattened form of query term *?- p(Z, h(Z,W), f(W))* is:
  - *X3 = h(X2,X5), X4 = f(X5), X1 = p(X2,X3,X4)*

Abstract Machine $\mathcal{M}_0$: Tokenized Form

- For each flattened term $Xi = f(Xi_1,\ldots,Xi_n)$, its *tokenized form* is the sequence of tokens $Xi = f/n, Xi_1, \ldots, Xi_n$.

- The tokenized form of query term *?- p(Z, h(Z,W), f(W))* is a stream of 9 tokens:
  - *X3 = h/2, X2, X5, X4 = f/1, X5, X1 = p/3, X2, X3, X4*

- There are three kinds of tokens to process:
  - A register associated with a structure functor
    X3 = h/2, *X2, X5,* ***X4 = f/1***, *X5,* ***X1 = p/3***, *X2, X3, X4*
  - A first-seen register in the stream
    X3 = h/2, ***X2, X5***, *X4 = f/1, X5, X1 = p/3, X2, X3, X4*
  - An already-seen register in the stream
    X3 = h/2, X2, X5, X4 = f/1, ***X5***, *X1 = p/3,* ***X2, X3, X4***

Abstract Machine $\mathcal{M}_0$: Compiling Queries

- 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. Respectively, each of the three kinds of tokens indicates a different action:
  - `put_structure f/n,Xi`

    push a new `STR` (and adjoining functor) cell onto the heap and copy that cell into the allocated register address.
  - `set_variable Xi`

    push a new `REF` cell onto the heap containing its own address, and copy it into the given register.
  - `set_value Xi`

    push a new cell onto the heap and copy into it the register's value.

Abstract Machine $\mathcal{M}_0$: Query Instructions

- Tokenized form of query term *?- p(Z, h(Z,W), f(W))*:
  - *X3 = h/2, X2, X5, X4 = f/1, X5, X1 = p/3, X2, X3, X4*
- Compiled code for query term *?- p(Z, h(Z,W), f(W))*:

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

Abstract Machine $\mathcal{M}_0$: Query Instructions

$\mathcal{M}_0$ uses a global heap register `H` to keep the address of the next free cell in the `HEAP`.

- `put_structure f/n,Xi`

```
HEAP[H] = <STR,H+1>
HEAP[H+1] = f/n
X[i] = HEAP[H]
H = H+2
```

- `set_variable Xi`

```
HEAP[H] = <REF,H>
X[i] = HEAP[H]
H = H+1
```

- `set_value Xi`

```
HEAP[H] = X[i]
H = H+1
```

Abstract Machine $\mathcal{M}_0$: Query Instructions

- Heap representation for query term *?- p(Z, h(Z,W), f(W))*.

X3	0	STR	1	put_structure h/2,X3
	1	*h/2*		
X2	2	REF	2	set_variable X2
X5	3	REF	3	set_variable X5
X4	4	STR	5	put_structure f/1,X4
	5	*f/1*		
	6	REF	3	set_value X5
X1	7	STR	8	put_structure p/3,X1
	8	*p/3*		
	9	REF	2	set_value X2
	10	STR	1	set_value X3
	11	STR	5	set_value X4

Abstract Machine $\mathcal{M}_0$: Compiling Programs

- Compiling a program term *p* assumes that a query term *?- q* has built a term on the heap and set register *X1* to contain its address.
- Code for *p* consists of:
  - Following the term structure already present in *X1* as long as it matches the term structure of *p*.
  - When an unbound `REF` cell is encountered, then it is bound to a new term that is built on the heap as an exemplar of the corresponding subterm in *p*.
  - Variable binding creates reference chains. Dereferencing is performed by a *deref()* procedure 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, returning the cell address.
- The code for an $\mathcal{L}_0$ program then 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.

Abstract Machine $\mathcal{M}_0$: Compiling Programs

- Variable registers *X1*, *X2*, ... are allocated as before. But now the flattened form follows a *top down* order because query data from the heap are assumed available.

- The variable registers allocated to program term *p(f(X), h(Y,f(a)),Y)* are:
  - *X1 = p(X2,X3,X4)*
  - *X2 = f(X5)*
  - *X3 = h(X4,X6)*
  - *X4 = Y*
  - *X5 = X*
  - *X6 = f(X7)*
  - *X7 = a*
- The flattened form of program term *p(f(X), h(Y,f(a)),Y)* is:
  - *X1 = p(X2,X3,X4), X2 = f(X5), X3 = h(X4,X6), X6 = f(X7), X7 = a*

Abstract Machine $\mathcal{M}_0$: Program Instructions

- Each flattened term $Xi = f(Xi_1,...,Xi_n)$ is tokenized as before as $Xi = f/n, Xi_1, ..., Xi_n$.

- The tokenized form of program term *p(f(X), h(Y,f(a)),Y)* is a stream of 12 tokens:
  - *X1 = p/3, X2, X3, X4, X2 = f/1, X5, X3 = h/2, X4, X6, X6 = f/1, X7, X7 = a/0*

- Again, there are three kinds of tokens to process:
  - A register associated with a structure functor – instruction `get_structure f/n,Xi`

    X1 = p/3, *X2*, *X3*, *X4*, ***X2 = f/1***, *X5*, ***X3 = h/2***, *X4*, *X6*, ***X6 = f/1***, *X7*, ***X7 = a/0***
  - A first-seen register in the stream – instruction `unify_variable Xi`

    X1 = p/3, ***X2***, ***X3***, ***X4***, *X2 = f/1*, ***X5***, *X3 = h/2*, *X4*, ***X6***, *X6 = f/1*, ***X7***, *X7 = a/0*
  - An already-seen register in the stream – instruction `unify_value Xi`

    X1 = p/3, *X2*, *X3*, *X4*, *X2 = f/1*, *X5*, *X3 = h/2*, ***X4***, *X6*, *X6 = f/1*, *X7*, *X7 = a/0*

Abstract Machine $\mathcal{M}_0$: Program Instructions

- Tokenized form of program term *p(f(X), h(Y,f(a)),Y)*:
  - *X1 = p/3, X2, X3, X4, X2 = f/1, X5, X3 = h/2, X4, X6, X6 = f/1, X7, X7 = a/0*
- Compiled code for program term *p(f(X), h(Y,f(a)),Y)*:

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

Abstract Machine $\mathcal{M}_0$: Read/Write Mode

- $\mathcal{M}_0$ uses a global subterm register `S` to keep the heap address of the next subterm to be matched in `READ` mode.
- Mode is set by instruction `get_structure f/n,Xi`:
  - if *deref(Xi)* returns a `REF` cell (unbound variable), then push a new `STR` cell pointing to *f/n* onto the heap, bind the `REF` cell to it and set mode to `WRITE`.
  - if *deref(Xi)* returns an `STR` cell pointing to *f/n*, then set register `S` to the heap address following that functor cell's and set mode to `READ`.
  - Otherwise, the program fails.

Abstract Machine $\mathcal{M}_0$: Program Instructions

- get_structure f/n,Xi

```
addr = deref(X[i])
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
                            S = a+1
                            mode = READ
                     else
                            fail()
        other:       fail()
```

Abstract Machine $\mathcal{M}_0$: Variable Binding

- The *bind()* procedure is performed on two store addresses, at least one of which is an unbound `REF` cell.
  - It binds the unbound `REF` cell to the other cell, i.e., it changes the data field of the unbound `REF` cell to contain the address of the other cell.
  - If both addresses are unbound `REF` cells, then the binding direction is chosen arbitrarily.

Abstract Machine $\mathcal{M}_0$: Read/Write Mode

- The unify instructions then depend on whether a term is to be matched from the heap (`READ` mode) or to be built on the heap (`WRITE` mode).
  - For matching, they seek to recognize data from the heap as those of the term at corresponding positions, proceeding if successful and failing otherwise.
  - For building, they work exactly like the set query instructions.
- `unify_variable Xi`
  - In `READ` mode, sets *Xi* to the contents of the heap at address `S`.
  - In `WRITE` mode, a new unbound `REF` cell is pushed onto 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 both modes, `S` is then incremented by one.

Abstract Machine $\mathcal{M}_0$: Program Instructions

- unify_variable Xi

```
case mode of
        READ:           X[i] = HEAP[S]
        WRITE:          HEAP[H] = <REF,H>
                        X[i] = HEAP[H]
                        H = H+1
S = S+1
```

- unify_value Xi

```
case mode of
        READ:           unify(X[i],S)
        WRITE:          HEAP[H] = X[i]
                        H = H+1
S = S+1
```

Abstract Machine $\mathcal{M}_0$: Unify Procedure

```
unify(address a1, address a2)
     push(a1,PDL)
     push(a2,PDL)
     while not_empty(PDL) do
             d1 = deref(pop(PDL))
             d2 = deref(pop(PDL))
             if (d1 != d2) then
                     <t1,v1> = STORE[d1]
                     <t2,v2> = STORE[d2]
                     if (t1 = REF or t2 = REF) then
                             bind(d1,d2)
                     else    // t1 = STR and t2 = STR
                             f1/n1 = STORE[v1]
                             f2/n2 = STORE[v2]
                             if (f1 = f2 and n1 = n2) then
                                     for i = 1 to n1 do
                                             push(v1+i,PDL)
                                             push(v2+i,PDL)
                             else fail()
```

Abstract Machine $\mathcal{M}_0$: Summary

- Global storage areas
  - `HEAP`: to represent terms
- Global registers
  - `Xi`: variable registers
  - `H`: heap register
  - `S`: subterm register
- Query instructions
  - `put_structure f/n,Xi`
  - `set_variable Xi`
  - `set_value Xi`
- Program instructions
  - `get_structure f/n,Xi`
  - `unify_variable Xi`
  - `unify_value Xi`

Language $\mathcal{L}_1$

Syntax

- Similar to $\mathcal{L}_0$, but now a program may be a *set of first-order atoms* p_1, …, p_n each defining *at most one fact per predicate name*.
- Language $\mathcal{L}_1$ makes now a distinction between *atoms* (terms whose functor is a predicate name) and *terms* (arguments to a predicate).

Semantics

- Execution of a query *?- q* connects to the appropriate predicate definition p_i, for computing the MGU of predicate *pi* and query *?- q*, or fails if none predicate definition exists for the query invoked.

Abstract Machine $\mathcal{M}_1$: Code Area

- $\mathcal{M}_1$ uses a global storage area called `CODE` where compiled code is stored.
- The code area is an array of possibly labeled instructions consisting of opcodes followed by operands. The size of an instruction stored at address `CODE[a]` is given by the expression `instruction_size(a)`.
- The standard execution order of instructions is sequential. $\mathcal{M}_1$ uses a global program register `P` to keep the address of the next instruction to execute. Unless failure occurs, most machine instructions are implicitly assumed, to increment `P` by `instruction_size(P)`.

Abstract Machine $\mathcal{M}_1$: Control Instructions

- 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.
- `call p/n`

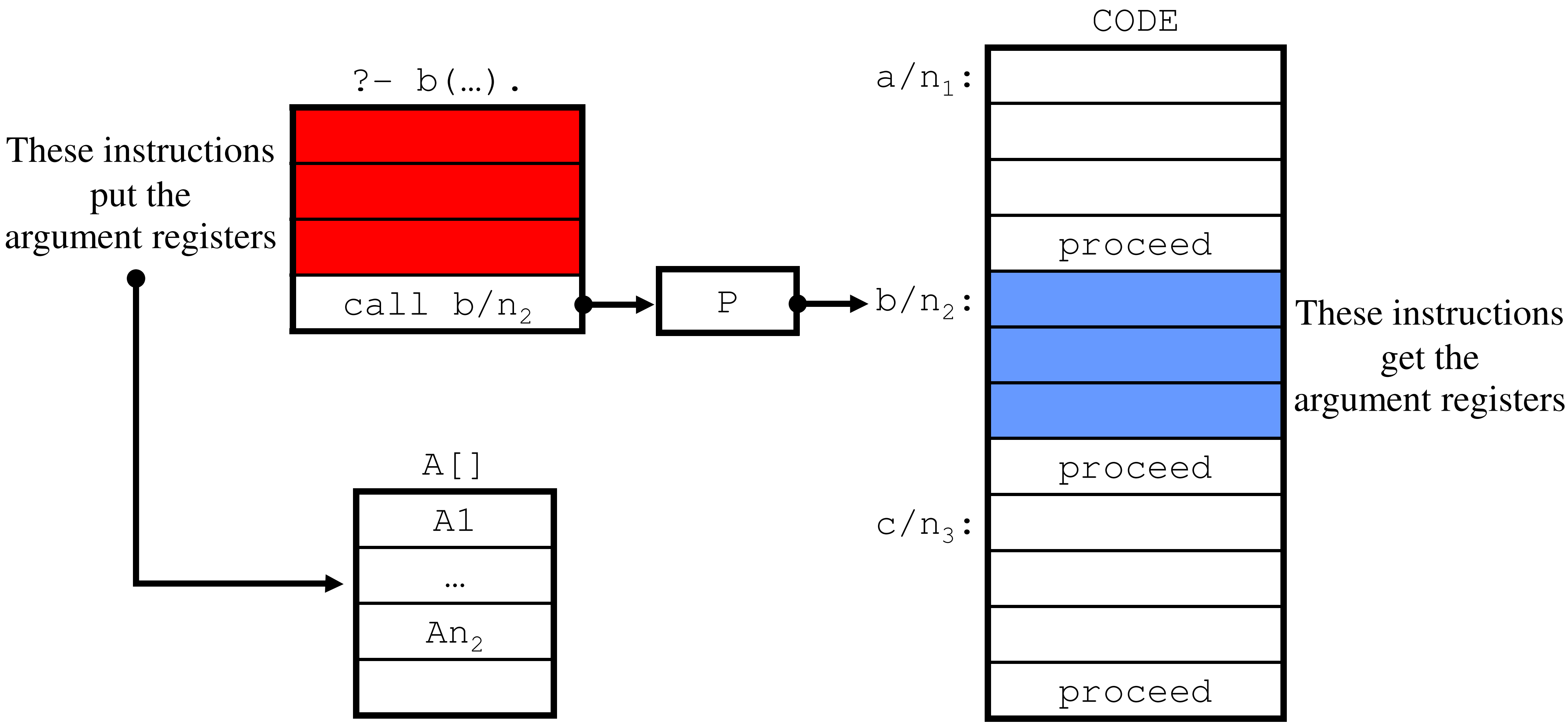
  Sets `P` to the address in the code area of instruction labeled *p/n*. If the procedure *p/n* is not defined, failure occurs and overall execution aborts. Labels are symbolic entry points into the code area used as operands of instructions for transferring control to the code labeled accordingly. Therefore, there is no need to store a procedure name in the heap as it denotes a key into a compiled instruction sequence.
- `proceed`

  Indicates the end of a fact's instruction sequence.

Abstract Machine $\mathcal{M}_1$: Argument Registers

- In $\mathcal{M}_1$, unification between fact and query terms amounts to solving, not one, but many equations, simultaneously.
- Registers *X1*, ..., *Xn* 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 *Ai* rather than *Xi* when the *i*-th register contains the *i*-th argument. Where register *Xi* is not used as an argument register, then it is written as usual.
- In $\mathcal{M}_1$, the *flattened form* of a query term is the *ordered* sequence of register assignments of the form $Ai = f(Xj_1,...,Xj_n)$, $Ai = Xj$ or $Xj = f(Xj_1,...,Xj_n)$ so that a variable register *Xj* is assigned *before* it is used as an argument in a subterm.

Abstract Machine $\mathcal{M}_1$: Argument Registers



Abstract Machine $\mathcal{M}_1$: Argument Registers

- The registers allocated to term *p(Z, h(Z,W), f(W))* are:
  - *A1 = X4*
  - *A2 = h(X4,X5)*
  - *A3 = f(X5)*
  - *X4 = Z*
  - *X5 = W*
- The flattened form of query term *?- p(Z, h(Z,W), f(W))* is:
  - *A1 = X4, A2 = h(X4,X5), A3 = f(X5)*
- The tokenized form of query term *?- p(Z, h(Z,W), f(W))* is:
  - *A1 = X4, A2 = h/2, X4, X5, A3 = f/1, X5*

Abstract Machine $\mathcal{M}_1$: Argument Registers

- The registers allocated to term *p(f(X), h(Y,f(a)), Y)* are:
  - *A1 = f(X4)*
  - *A2 = h(X5,X6)*
  - *A3 = X5*
  - *X4 = X*
  - *X5 = Y*
  - *X6 = f(X7)*
  - *X7 = a*
- The flattened form of program term *p(f(X), h(Y,f(a)), Y)* is:
  - *A1 = f(X4), A2 = h(X5,X6), A3 = X5, X6 = f(X7), X7 = a*
- The tokenized form of program term *p(f(X), h(Y,f(a)), Y)* is:
  - *A1 = f/1, X4, A2 = h/2, X5, X6, A3 = X5, X6 = f/1, X7, X7 = a/0*

Abstract Machine $\mathcal{M}_1$: Argument Instructions

- The argument instructions are needed in $\mathcal{M}_1$ to handle variable registers that appear in argument positions.
- **In a query**
  - A first-seen variable register *Xj* appearing in the *i*-th argument position pushes a new unbound `REF` cell onto the heap and copies it into *Xj* as well as argument register *Ai*.
  - An already-seen variable register *Xj* appearing in the *i*-th argument position copies its value into argument register *Ai*.
- **In a program fact**
  - A first-seen variable register *Xj* appearing in the *i*-th argument position sets it to the value of argument register *Ai*.
  - An already-seen variable register *Xj* appearing in the *i*-th argument position unifies it with the value of *Ai*.

Abstract Machine $\mathcal{M}_1$: Argument Instructions

- `put_variable Xn,Ai`
```
HEAP[H] = <REF,H>
X[n] = HEAP[H]
A[i] = HEAP[H]
H = H+1
```
- `put_value Xn,Ai`
```
A[i] = X[n]
```
- `get_variable Xn,Ai`
```
X[n] = A[i]
```
- `get_value Xn,Ai`
```
unify(X[n],A[i])
```

Abstract Machine $\mathcal{M}_1$: Argument Instructions

- Tokenized form of query term *?- p(Z, h(Z,W), f(W))*:
  - *A1 = X4, A2 = h/2, X4, X5, A3 = f/1, X5*
- Compiled code for query term *?- p(Z, h(Z,W), f(W))*:

```
put_variable X4,A1        % A1 = X4
put_structure h/2,A2      % A2 = h
set_value X4              %        (X4,
set_variable X5           %            X5)
put_structure f/1,A3      % A3 = f
set_value X5              %        (X5)
call p/3                  % p(A1,A2,A3)
```

Abstract Machine $\mathcal{M}_1$: Argument Instructions

- Tokenized form of program term *p(f(X), h(Y,f(a)), Y)*:
  - *A1 = f/1, X4, A2 = h/2, X5, X6, A3 = X5, X6 = f/1, X7, X7 = a/0*
- Compiled code for program term *p(f(X), h(Y,f(a)),Y)*:

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

Abstract Machine $\mathcal{M}_1$: Summary

- Global storage areas
  - `CODE`: to store compiled code
  - `HEAP`: to represent terms
- Global registers
  - `Ai`: argument registers
  - `Xi`: variable registers
  - `P`: program register
  - `H`: heap register
  - `S`: subterm register

Abstract Machine $\mathcal{M}_1$: Summary

- Query instructions
  - `put_structure f/n,Xi`
  - `set_variable Xi`
  - `set_value Xi`
  - `put_structure f/n,Ai`
  - `put_variable Xn,Ai`
  - `put_value Xn,Ai`
  - `call p/n`
- Program instructions
  - `get_structure f/n,Xi`
  - `unify_variable Xi`
  - `unify_value Xi`
  - `get_structure f/n,Ai`
  - `get_variable Xn,Ai`
  - `get_value Xn,Ai`
  - `proceed`

Language $\mathcal{L}_2$

Syntax

- Similar to $\mathcal{L}_1$, but now a program is a *set of predicates of the form* p*:-* $b_1, \ldots, b_n$ where p and the b_i's are atoms defining *at most one clause per predicate name*.
- Predicates are no longer reduced only to facts but may also have bodies. A body is a conjunctive sequence of atoms (or *goals*). When $n = 0$, the clause is called a *fact* and written without the implication symbol (:-). When $n > 0$, the clause is called a *rule*, atom p is called the *head of the rule* and the b_i's are called the *body of the rule*.

Semantics

- A $\mathcal{L}_2$ query is now a sequence of atoms (or *goals*) of the form *?-* $q_1, \ldots, q_k$.
- Execution of such a query *?-* $q_1, \ldots, q_k$ consists of repeated application of *leftmost resolution* until the empty query, or failure, is obtained.

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

- Always unify the leftmost query goal with its definition's head or fail if none exists.
- If unification succeeds, replace the query goal by its definition's body, variables in scope bearing the binding side-effects of unification.
- Therefore, executing a query in $\mathcal{L}_2$ either:
  - Terminates with success (the result is the bindings of the variables in the query).
  - Terminates with failure.
  - Never terminates.

Abstract Machine $\mathcal{M}_2$: Compiling Goals

- To compile a rule body or a query with several goals, we can concatenate $\mathcal{M}_1$'s compiled code for each goal. However, we must take special care with:
  - Continuing the execution of a goal sequence.
  - Avoiding conflicts in the use of argument registers.

- Compiled pseudo-code for clause $p_0(\ldots)$:- $p_1(\ldots), \ldots, p_n(\ldots)$:

```
'get arguments of p0'          % not needed for queries
'put arguments of p1'
call p1
...
'put arguments of pn'
call pn
```

Abstract Machine $\mathcal{M}_2$: Control Instructions

- After successfully returning from a call to a fact, now `proceed` must continue execution back to the instruction in the goal sequence following the call.
- $\mathcal{M}_2$ uses a global continuation point register `CP` to save and restore the address of the next instruction to follow up with upon successful return from a call and alters $\mathcal{M}_1$'s control instructions to:
  - `call p/n`
    ```
    CP = P + instruction_size(P)
    P = @(p/n)
    ```
  - `proceed`
    ```
    P = CP
    ```
- As before, when the procedure *p/n* is not defined, execution fails.

Abstract Machine $\mathcal{M}_2$: Permanent Variables

- Variables which occur in more than one body goal are called *permanent variables* as they have to outlive the call where they first appear. All other variables in a scope that are not permanent are called *temporary variables*. We write a permanent variable as *Yi*, and use *Xi* as before for temporary variables.
- 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.*, in the example below *X* is temporary).

- **Problem:** because the same variable registers are used by every body goal, permanent variables run the risk of being overwritten by intervening goals. For example, in rule

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

  no guarantee can be made that the variables *Y* and *Z* are still in registers after executing *q*.
- **Solution:** save permanent variables in an *environment* associated with each activation of the call they appear in.

Abstract Machine $\mathcal{M}_2$: Environments

- $\mathcal{M}_2$ uses a global storage area called `STACK` to store environments and a global environment register `E` to keep the address of the latest environment on `STACK`.
- An environment is pushed onto `STACK` upon a non-fact entry call, and popped from `STACK` upon return. Environments are used to save the permanent variables and the continuation point.
- The `STACK` is organized as a linked list of environment 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

Abstract Machine $\mathcal{M}_2$: Environment Instructions

- `allocate N`

  Creates and pushes an environment frame for N permanent variables onto `STACK`.
- `deallocate`

  Discards the environment frame on top of `STACK` and sets execution to continue at the continuation point recovered from the environment being discarded.

- Compiled pseudo-code for clause $p_0(\ldots)$:- $p_1(\ldots), \ldots, p_n(\ldots)$:

```
allocate N
'get arguments of p0'                    % not needed for queries
'put arguments of p1'
call p1
...
'put arguments of pn'
call pn
deallocate
```

Abstract Machine $\mathcal{M}_2$: Environment Instructions

- `allocate N`

```
newE = E + STACK[E+2] + 3
STACK[newE] = E
STACK[newE+1] = CP
STACK[newE+2] = N
E = newE
P = P + instruction_size(P)
```

- `deallocate`

```
P = STACK[E+1]
E = STACK[E]
```

Abstract Machine $\mathcal{M}_2$: Environment Instructions

- The variable registers allocated to clause *p(X,Y) :- q(X,Z), r(Z,Y)* are:
  - *X3 = X, Y1 = Y, Y2 = Z*
- Compiled code for clause *p(X,Y) :- q(X,Z), r(Z,Y)*:

```
p/2: allocate 2              %
     get_variable X3,A1      % A1 = X3
     get_variable Y1,A2      % A2 = Y1
     put_value X3,A1         % A1 = X3
     put_variable Y2,A2      % A2 = Y2
     call q/2                % q(A1,A2)
     put_value Y2,A1         % A1 = Y2
     put_value Y1,A2         % A2 = Y1
     call r/2                % r(A1,A2)
     deallocate              %
```

Abstract Machine $\mathcal{M}_2$: Summary

- Global storage areas
  - CODE: to store compiled code
  - HEAP: to represent terms
  - STACK: to store environments
- Global registers
  - Ai: argument registers
  - Xi: temporary variable registers
  - Yi: permanent variable registers
  - P: program register
  - CP: continuation point register
  - H: heap register
  - S: subterm register
  - E: environment register

Abstract Machine $\mathcal{M}_2$: Summary

- Query instructions
  - `put_structure f/n,Xi`
  - `set_variable Xi`
  - `set_value Xi`
  - `put_structure f/n,Ai`
  - `put_variable Xn,Ai`
  - `put_value Xn,Ai`
  - `call p/n`
  - `allocate N`
  - `deallocate`

Abstract Machine $\mathcal{M}_2$: Summary

- Program instructions
  - `get_structure f/n,Xi`
  - `unify_variable Xi`
  - `unify_value Xi`
  - `get_structure f/n,Ai`
  - `get_variable Xn,Ai`
  - `get_value Xn,Ai`
  - `proceed`
  - `allocate N`
  - `deallocate`

Language $\mathcal{L}_3$

Syntax

- $\mathcal{L}_3$ extends $\mathcal{L}_2$ to allow *disjunctive definitions of a predicate*.
- A predicate definition is an ordered sequence of clauses (facts and/or rules) consisting of all and only those whose head atoms share the same predicate name.

Semantics

- $\mathcal{L}_3$ queries are the same as those of $\mathcal{L}_2$. Query execution 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 point* at the moment of failure is reexamined first.

Abstract Machine $\mathcal{M}_3$: Choice Points

- $\mathcal{M}_3$ now saves the state of computation at each procedure call offering alternatives. We call such a state a *choice point*. A choice point 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.
- A choice point is created by the first alternative of a predicate defined by more than one alternative (if a predicate contains only one clause, there is no need to create a choice point). Then, it is updated (with the alternative to try next) by intermediate (but non ultimate) alternatives. Finally, it is discarded by the last alternative.
- $\mathcal{M}_3$ 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*.

Abstract Machine $\mathcal{M}_3$: Choice Points

- $\mathcal{M}_3$ uses a global backtrack register `B` to keep the address of the latest choice point.
- Upon failure, computation is resumed from the state recovered from the choice point indicated by `B`. If the choice point offers no more alternatives, it is popped off the OR-stack by resetting `B` to its predecessor if one exists. Otherwise, the computation fails terminally and execution aborts.

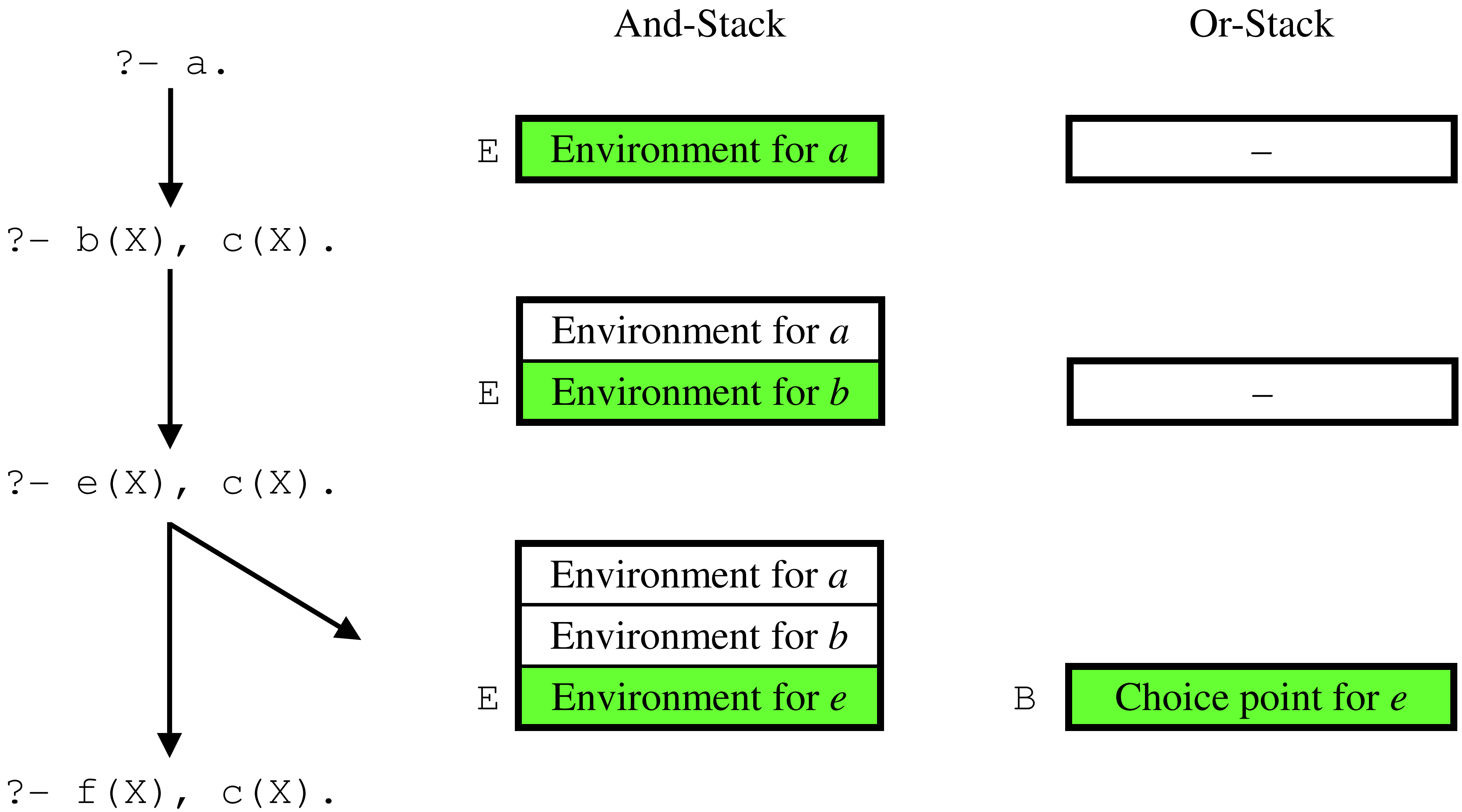
Abstract Machine $\mathcal{M}_3$: Environment Protection

- **Problem:** in $\mathcal{M}_2$, it is safe to deallocate an environment frame at the end of a rule. However, in $\mathcal{M}_3$ this is no longer true: failure may force reconsidering a choice point from a computation in the middle of a rule whose environment has been deallocated.

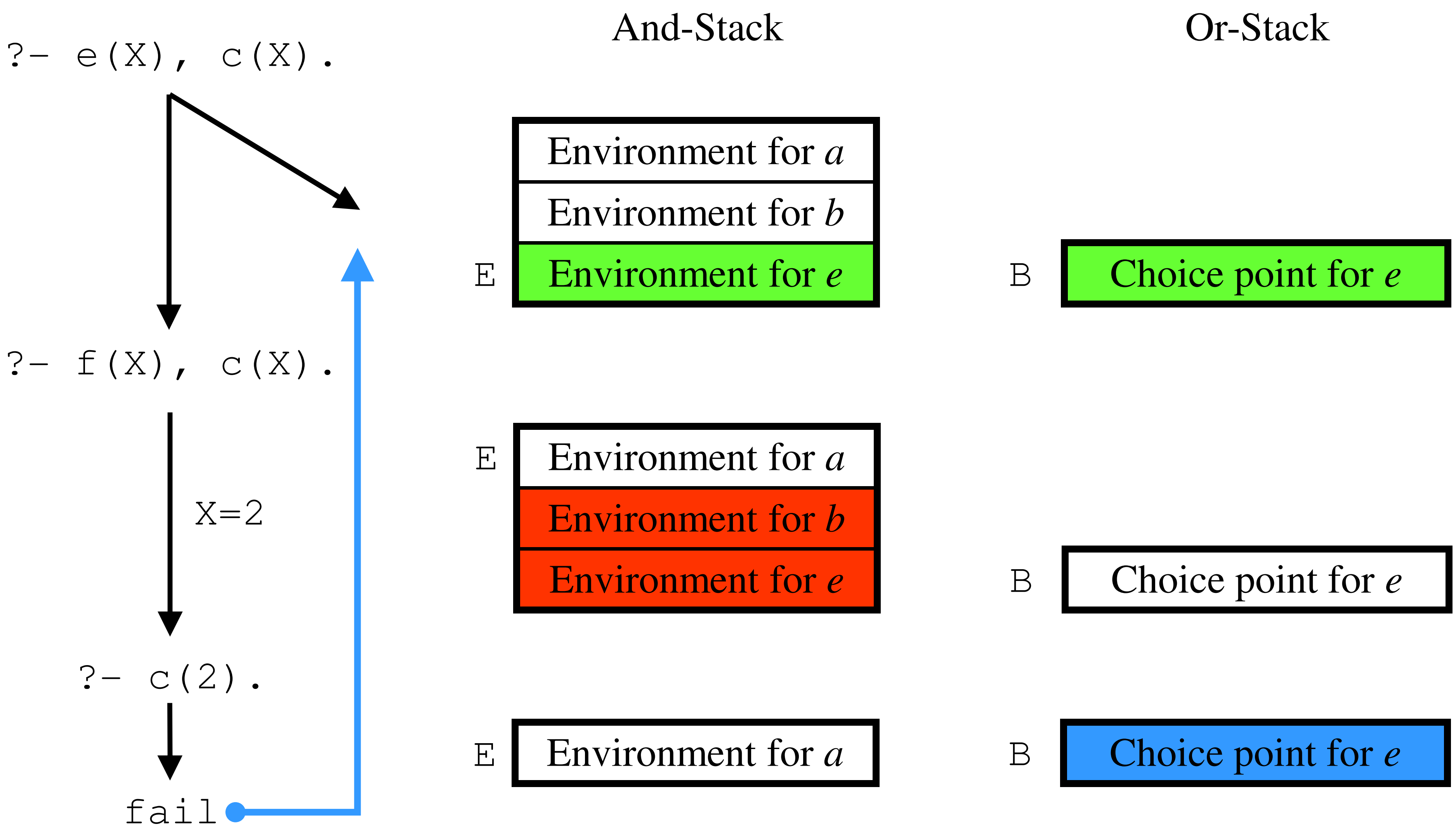
- **Example:** consider the following program and the query *?- a*:

```
a:- b(X), c(X).
b(X):- e(X).
c(1).
e(X):- f(X).
e(X):- g(X).
f(2).
g(1).
```

Abstract Machine $\mathcal{M}_3$: Environment Protection



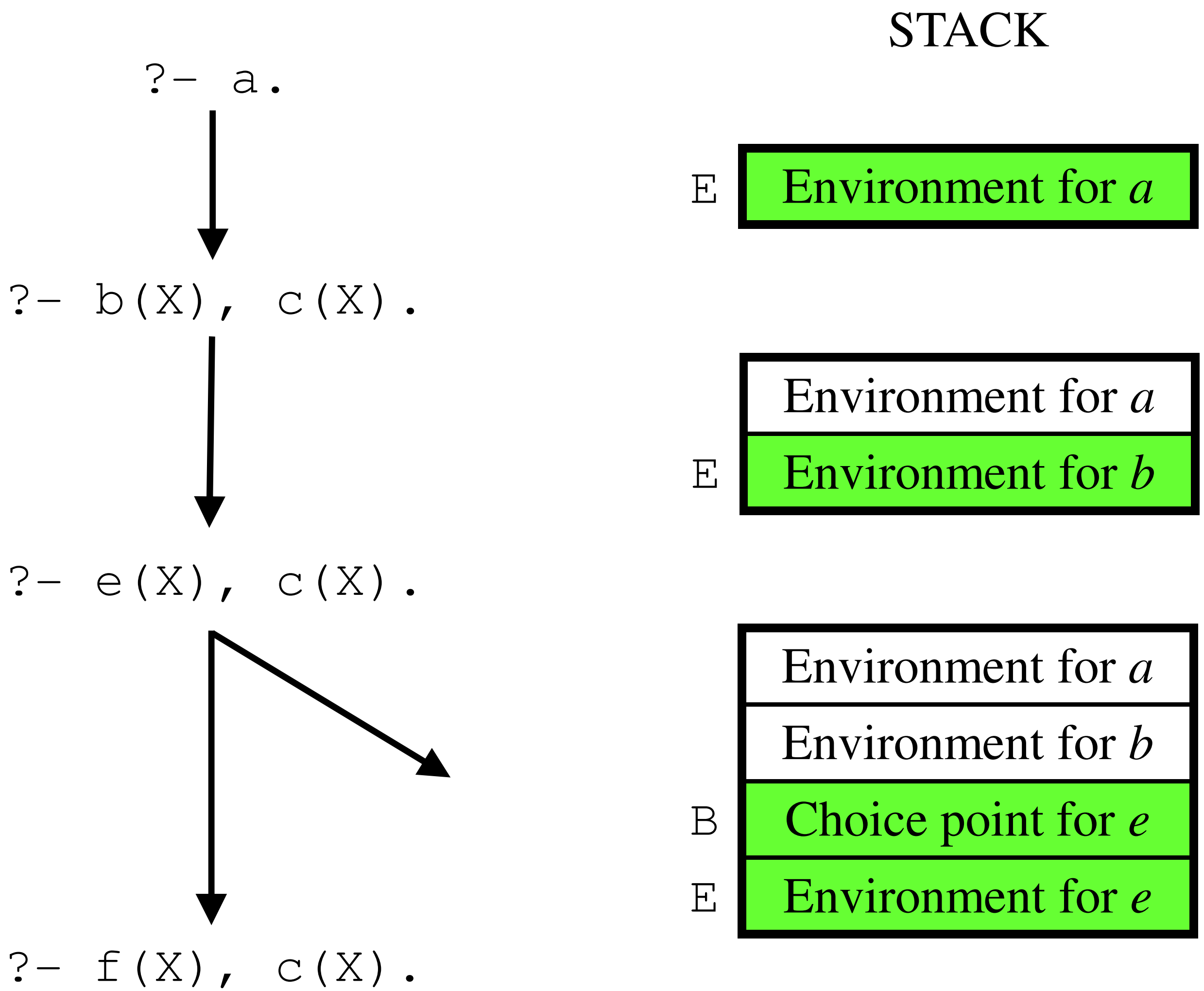
Abstract Machine $\mathcal{M}_3$: Environment Protection



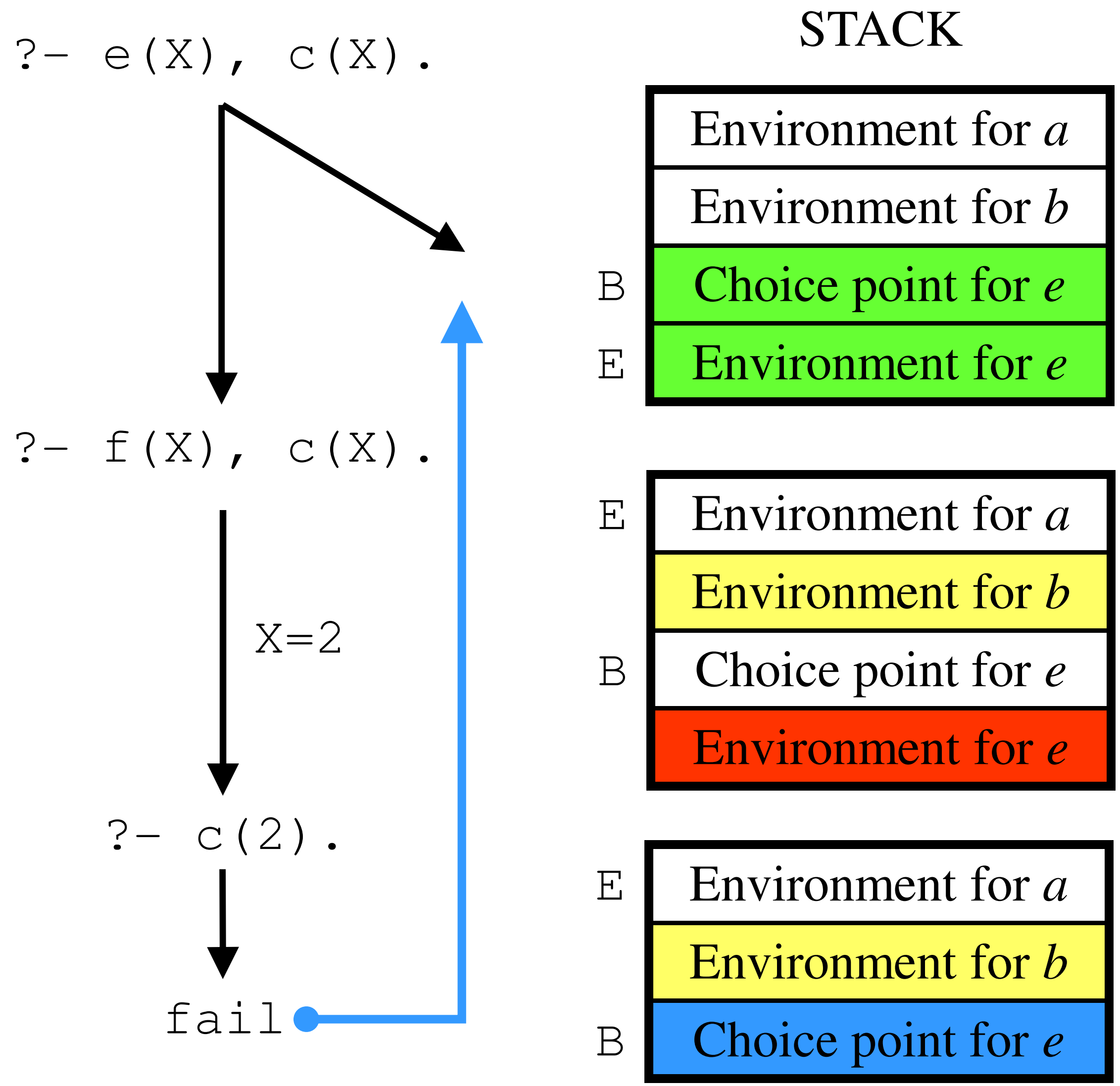
Abstract Machine $\mathcal{M}_3$: Environment Protection

- 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.
- **Solution:** every choice point must *protect* from deallocation all environment frames existing before its creation. To do that $\mathcal{M}_3$ uses the same stack for *both* environments and choice points:
  - As long as a choice point is active, it forces allocation of further environments on top of it, avoiding overwriting the (even explicitly deallocated) older environments.
  - Safe resurrection of a deallocated protected environment is automatic when coming back to an alternative from the 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.

Abstract Machine $\mathcal{M}_3$: Environment Protection

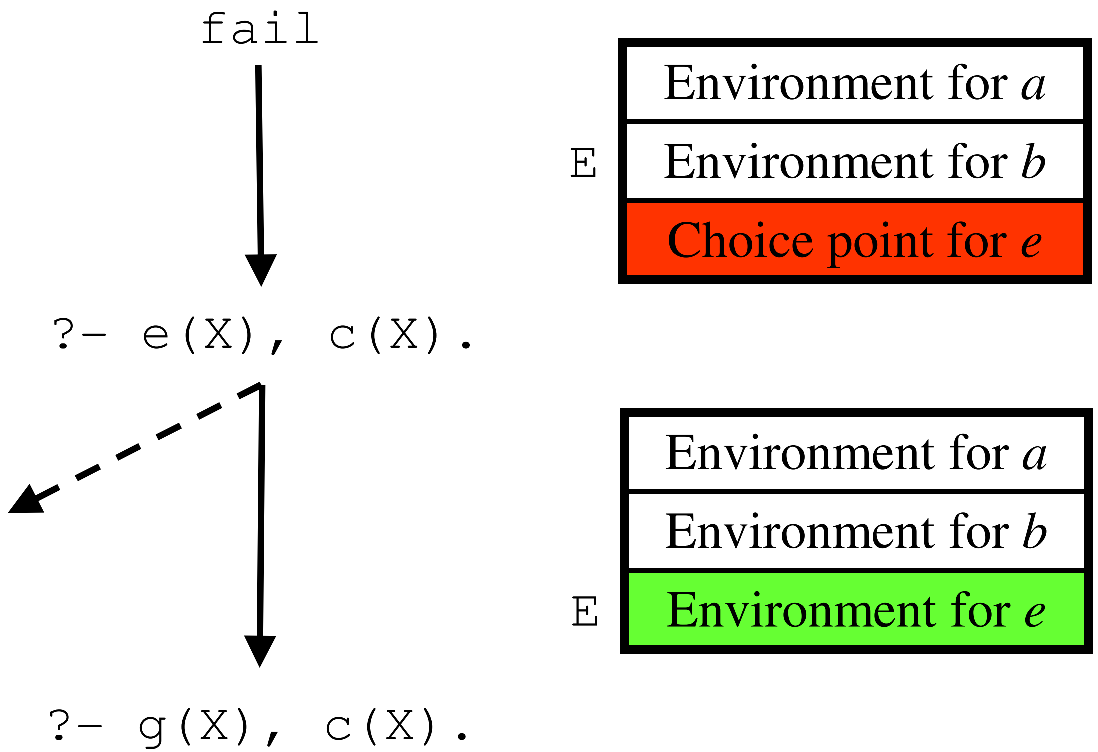


Abstract Machine $\mathcal{M}_3$: Environment Protection



Abstract Machine $\mathcal{M}_3$: Environment Protection

- 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 that of *b*, the one current at the time of this choice point's creation.



Abstract Machine $\mathcal{M}_3$: Undoing Bindings

- Binding effects must be undone when reconsidering a choice point.
- $\mathcal{M}_3$ uses a global storage area called `TRAIL` to record all variables which need to be *reset to unbound upon backtracking*, a global trail register `TR` to keep the address of the next free cell in the `TRAIL` and a global heap backtrack register `HB` to keep the value of `H` at the time of the latest choice point’s creation.
- Only *conditional* bindings need to be trailed. A conditional binding is one affecting a variable existing before creation of the current choice point:
  - `HEAP[a]` is conditional *iff* `a < HB`.
  - `STACK[a]` is conditional *iff* `a < B`.

Abstract Machine $\mathcal{M}_3$: What's in a Choice Point

- The argument registers *A1, ..., An* of the predicate being called.
- The current environment (register `E`) to recover as a protected environment.
- The continuation point (register `CP`) as the current choice will overwrite it.
- The latest choice point (register `B`) where to backtrack in case all alternatives offered by the current choice point fail.
- The next clause to try for this predicate 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 (register `TR`) which is needed as the boundary where to unwind the trail upon backtracking.
- The current top of heap (register `H`) which is needed to recover (garbage) heap space of all the structures and variables constructed during the failed attempt.

Abstract Machine $\mathcal{M}_3$: What's in a Choice Point

B	n	number of arguments
B+1	A1	argument register 1
...	...	
B+n	An	argument register n
B+n+1	E	current environment
B+n+2	CP	continuation point
B+n+3	B	previous choice point
B+n+4	BP	next clause to try
B+n+5	TR	current trail pointer
B+n+6	H	current heap pointer

Abstract Machine $\mathcal{M}_3$: Backtracking

- All instructions where failure may occur (unification instructions and predicate calls) now backtrack to the next clause to try for the current choice point, if one exists. Otherwise, the computation fails terminally and execution aborts.
- `fail()`

```
if (B != NULL)
      P = STACK[B+STACK[B]+4]
else
      abort()
```

Abstract Machine $\mathcal{M}_3$: Environment Instructions

- `allocate N`

```
if (E > B) then
        newE = E + STACK[E+2] + 3
else
        newE = B + STACK[B] + 7
STACK[newE] = E
STACK[newE+1] = CP
STACK[newE+2] = N
E = newE
P = P + instruction_size(P)
```

Abstract Machine $\mathcal{M}_3$: Choice Point Instructions

- $\mathcal{M}_3$ uses three different instructions to deal with multiple clause predicates.
- `try_me_else L`

  For the first clause. Allocates a new choice point setting its next clause field to `L` and the other fields according to the current context, and sets `B` to point to it.
- `retry_me_else L`

  For intermediate (but not ultimate) clauses. Resets all the necessary information from the current choice point and updates its next clause field to `L`.
- `trust_me`

  For the last clause. Resets all the necessary information from the current choice point and then discards it by resetting `B` to the value of its predecessor slot.

Abstract Machine $\mathcal{M}_3$: Choice Point Instructions

- For a two clause definition for a predicate *p/n* the pattern is:

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

- For more than two clauses the pattern is:

```
p/n: try_me_else L1
       'code for first clause'
 L1: retry_me_else L2
       'code for second clause'
       ...
Lk-1: retry_me_else Lk
       'code for penultimate clause'
 Lk: trust_me
       'code for last clause'
```

Abstract Machine $\mathcal{M}_3$: Choice Point Instructions

- Consider the following predicate defined by three clauses:

```
p(X,a).
p(b,X).
p(X,Y):- p(X,a), p(b,Y).
```

- Compiled code for clause *p(X,a)*:

```
p/2: try_me_else L1            %
     get_variable X3,A1        % A1 = X3
     get_structure a/0,A2      % A2 = a
     proceed                   %
```

- Compiled code for clause *p(b,X)*:

```
 L1: retry_me_else L2          %
     get_structure b/0,A1      % A1 = b
     get_variable X3,A2        % A2 = X3
     proceed                   %
```

Abstract Machine $\mathcal{M}_3$: Choice Point Instructions

- Compiled code for clause *p(X,Y):- p(X,a), p(b,Y)*:

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

Abstract Machine $\mathcal{M}_3$: Summary

- Global storage areas
  - `CODE`: to store compiled code
  - `HEAP`: to represent terms
  - `STACK`: to store environments and choice points
  - `TRAIL`: to record variables which need to be reset upon backtracking

Abstract Machine $\mathcal{M}_3$: Summary

- Global registers
  - Ai: argument registers
  - Xi: temporary variable registers
  - Yi: permanent variable registers
  - P: program register
  - CP: continuation point register
  - H: heap register
  - S: subterm register
  - E: environment register
  - B: backtrack register
  - HB: heap backtrack register
  - TR: trail register

Abstract Machine $\mathcal{M}_3$: Summary

- Query instructions
  - `put_structure f/n,Xi`
  - `set_variable Xi`
  - `set_value Xi`
  - `put_structure f/n,Ai`
  - `put_variable Xn,Ai`
  - `put_value Xn,Ai`
  - `call p/n`
  - `allocate N`
  - `deallocate`

Abstract Machine $\mathcal{M}_3$: Summary

- Program instructions
  - `get_structure f/n,Xi`
  - `unify_variable Xi`
  - `unify_value Xi`
  - `get_structure f/n,Ai`
  - `get_variable Xn,Ai`
  - `get_value Xn,Ai`
  - `proceed`
  - `allocate N`
  - `deallocate`
  - `try_me_else L`
  - `retry_me_else L`
  - `trust_me`

Optimizing the Design: WAM Principles

- **WAM Principle I**

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

- **WAM Principle II**

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

- **WAM Principle III**

  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.

Optimizing the Design: Heap Representation

- There is actually no need to allocate a systematic `STR` cell before each functor cell. 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

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

Optimizing the Design: Heap Representation

- However, all references from registers to functor cells should remain as structure cells. For this, we need to change the `put_structure` instruction from:

```
put_structure f/n,Xi
    HEAP[H] = <STR,H+1>
    HEAP[H+1] = f/n
    X[i] = HEAP[H]
    H = H+2
```

to:

```
put_structure f/n,Xi
    HEAP[H] = f/n
    X[i] = <STR,H>
    H = H+1
```

Optimizing the Design: Constants

- There are sequences of instructions that bind a register to a constant and then proceed pushing a cell onto the heap with the register's value:

```
put_structure c/0,Xi
...
set_value Xi
```

This sequences can be simplified into one specialized instruction:

```
set_constant c
```

- There are other sequences that bind a register and then proceed checking the presence of a constant on the heap:

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

This sequences can be simplified into one specialized instruction:

```
unify_constant c
```

Optimizing the Design: Constants

- Heap space for constants can also be saved when loading an argument register with a constant or binding an argument register to a constant. These operations can be simplified from those of structures to deal specifically with constants:

```
put_constant c,Ai
get_constant c,Ai
```

- To represent constants, we need a new sort of data cell tagged `CON`. For example, a heap representation starting at address 10 for term *f(b,g(a))* could be:

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

Optimizing the Design: Constants

- `put_constant c,Ai`

```
    A[i] = <CON,c>
```

- `set_constant c`

```
    HEAP[H] = <CON,c>
    H = H+1;
```

- `get_constant c,Ai`

```
    addr = deref(A[i])
    case STORE[addr] of
          <REF,_>:      STORE[addr] = <CON,c>
                        trail(addr)
          <CON,c'>:     if (c != c') then fail()
          other:        fail()
```

Optimizing the Design: Constants

- unify_constant c

```
case mode of
        READ:  addr = deref(S)
               case STORE[addr] of
                     <REF,_>:      STORE[addr] = <CON,c>
                                   trail(addr)
                     <CON,c'>:     if (c != c') then fail()
                     other:        fail()
        WRITE: HEAP[H] = <CON,c>
               H = H+1
```

Optimizing the Design: Lists

- Similarly to constants, heap space for non-empty lists can be saved when loading or binding a register to it. To represent lists we use a new sort of data cell tagged <`LIS`,*a*> indicating that cell *a* contains the heap address of the first element of a list pair.
- `put_list Xi`

```
X[i] = <LIS,H>
```

- `get_list Xi`

```
addr = deref(X[i])
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()
```

Optimizing the Design: Constant/List Instructions

- Flattened form of query *?- p(Z, [Z,W], f(W))*:
  - *X5 = [X6 / []], A1 = X4, A2 = [X4 / X5], A3 = f(X6)*
- Compiled code for query *?- p(Z, [Z,W], f(W))*:

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

Optimizing the Design: Constant/List Instructions

- Flattened form of clause *p(f(X), [Y,f(a)], Y)*:
  - *A1 = f(X4), A2 = [X5 / X6], A3 = X5, X6 = [X7 / [ ]], X7 = f(a)*
- Compiled code for clause *p(f(X), [Y,f(a)], Y)*:

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

Optimizing the Design: Anonymous Variables

- A single-occurrence variable in a non-argument position is called an *anonymous variable*. Anonymous variables need no registers and if many occur in a row as in *f(_,_,_)* they can be all processed in a single instruction.
- `set_void N`
  - Pushes N new unbound `REF` cells on the heap.
- `unify_void N`
  - In `WRITE` mode, pushes N new unbound `REF` cells on the heap.
  - In `READ` mode, skips the next N heap cells starting at address `S`.
- A single-occurrence variable in an argument position can be simply ignored.

Optimizing the Design: Anonymous Variables

- set_void N

```
for i = H to H+n-1 do
        HEAP[i] = <REF,i>
H = H+n
```

- unify_void N

```
case mode of
        READ:           S = S+n
        WRITE:          for i = H to H+n-1 do
                                HEAP[i] = <REF,i>
                        H = H+n
```

Optimizing the Design: Anonymous Variables

- Flattened form of clause *p(_, g(X), f(_,Y,_))*:
  - *A2 = g(_), A3 = f(_,_,_)*
- Compiled code for clause *p(_, g(X), f(_,Y,_))*:

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

Optimizing the Design: Better Register Allocation

- Some sequences of register instructions are vacuous operations and can be eliminated:

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

- Clever register allocation can also be used to allow peep-hole optimizations. Consider, for example, the code for clause *conc([ ], L, L)*:

```
conc/3: get_constant [],A1          % A1 = []
        get_variable X4,A2          % A2 = X4
        get_value X4,A3             % A3 = X4
        proceed                     %
```

- It is silly to use an extra register `X4` to unify `A2` with `A3`:

```
conc/3: get_constant [],A1          % A1 = []
        get_value A2,A3             % A3 = A2
        proceed                     %
```

Optimizing the Design: Register Allocation

- The variable registers allocated to clause *p(X,Y) :- q(X,Z), r(Z,Y)* are:
  - *X3 = X, Y1 = Y, Y2 = Z*
- Compiled code for clause *p(X,Y) :- q(X,Z), r(Z,Y)*:

```
p/2: allocate 2             %
     get_variable X3,A1     % A1 = X3
     get_variable Y1,A2     % A2 = Y1
     put_value X3,A1        % A1 = X3
     put_variable Y2,A2     % A2 = Y2
     call q/2               % q(A1,A2)
     put_value Y2,A1        % A1 = Y2
     put_value Y1,A2        % A2 = Y1
     call r/2               % r(A1,A2)
     deallocate             %
```

Optimizing the Design: Better Register Allocation

- A better variable register allocation for clause *p(X,Y) :- q(X,Z), r(Z,Y)* is:
  - *Y1 = Y, Y2 = Z*
- Better compiled code for clause *p(X,Y) :- q(X,Z), r(Z,Y)*:

```
p/2: allocate 2                  %
     get_variable Y1,A2          % A2 = Y1
     put_variable Y2,A2          % A2 = Y2
     call q/2                    % q(A1,A2)
     put_value Y2,A1             % A1 = Y2
     put_value Y1,A2             % A2 = Y1
     call r/2                    % r(A1,A2)
     deallocate                  %
```

Optimizing the Design: Last Call Optimization

- **Idea:** permanent variables are no longer needed after the `put` instructions preceding the last `call` instruction in the body. The current environment can thus be discarded *before* the last call in a rule's body.
- **Solution:** swap the `call/deallocate` sequence that always conclude a rule's compiled code into a `deallocate/call` sequence.

Optimizing the Design: Last Call Optimization

- **Caution I:** as `deallocate` is no longer the last instruction, it must reset `CP` rather than `P`. For this, we need to change the `deallocate` instruction from:

```
deallocate
    P = STACK[E+1]
    E = STACK[E]
```

to

```
deallocate
    CP = STACK[E+1]
    E = STACK[E]
    P = P + instruction_size(P)
```

Optimizing the Design: Last Call Optimization

- **Caution II:** but now when `call` is the last instruction, it must not set `CP` but only `P`. So, as we cannot modify `call`, since it is correct when not last:

```
call p/n
    CP = P + instruction_size(P)
    P = @(p/n)
```

we use a new instruction for last calls:

```
execute p/n
    P = @(p/n)
```

Optimizing the Design: Last Call Optimization

- Compiled code for clause *p(X,Y) :- q(X,Z), r(Z,Y)* with last call optimization:

```
p/2: allocate 2                 %
     get_variable Y1,A2         % A2 = Y1
     put_variable Y2,A2         % A2 = Y2
     call q/2                   % q(A1,A2)
     put_value Y2,A1            % A1 = Y2
     put_value Y1,A2            % A2 = Y1
     deallocate                 %
     execute r/2                % r(A1,A2)
```

Optimizing the Design: Chain Rules

- Applying last call optimization to a chain rule of the form *p(. . .) :- q(. . .)* results in:

```
p/n: allocate N
     'get arguments of p/n'
     'put arguments of q/m'
     deallocate
     execute q/m
```

- As all variables in a chain rule are necessarily temporary, with last call optimization the `allocate/deallocate` instructions became *useless* and can be eliminated:

```
p/n: 'get arguments of p/n'
     'put arguments of q/m'
     execute q/m
```

Optimizing the Design: Indexing

- To seed up clause selection, the WAM uses the *first argument as an indexing key*.
- However, a clause whose head has a variable key creates a search bottleneck. The clauses defining a predicate *p/n* are thus partitioned in subsequences *S1, …, Sm* such that each *Si* is:
  - A *single* clause with a variable key.
  - A *maximal subsequence* of contiguous clauses whose keys are not variables.
- These subsequences are then compiled using the usual choice point instructions:

```
p/n: try_me_else S2
     'code for subsequence S1'
 S2: retry_me_else S3
     'code for subsequence S2'
     ...
 Sm: trust me
     'code for subsequence Sm'
```

Optimizing the Design: Indexing

- Consider the following definition for predicate *call/1*:

```
call(or(X,Y)):- call(X).
call(trace):- trace.
call(or(X,Y)):- call(Y).
call(notrace):- notrace.
call(nl):- nl.
call(X):- builtin(X).
call(X):- extern(X).
call(call(X)):- call(X).
call(repeat).
call(repeat):- call(repeat).
call(true).
```

- Using the first argument as an indexing key, into how many subsequences predicate *call/1* can be partitioned?

Optimizing the Design: Indexing

- Subsequence *S1* for predicate *call/1*:

```
call(or(X,Y)):- call(X).
call(trace):- trace.
call(or(X,Y)):- call(Y).
call(notrace):- notrace.
call(nl):- nl.
```

- Subsequence *S2* for predicate *call/1*:

```
call(X):- builtin(X).
```

- Subsequence *S3* for predicate *call/1*:

```
call(X):- extern(X).
```

- Subsequence *S4* for predicate *call/1*:

```
call(call(X)):- call(X).
call(repeat).
call(repeat):- call(repeat).
call(true).
```

Optimizing the Design: Indexing

- Compiled code for predicate *call/1*:

```
call/1: try_me_else S2
         ‘indexing code for subsequence S1’
    S2: retry_me_else S3
        execute builtin/1
    S3: retry_me_else S4
        execute extern/1
    S4: trust_me
         ‘indexing code for subsequence S4’
```

Optimizing the Design: Indexing

- The general indexing code pattern is:

  first level indexing

  second level indexing

  third level indexing

  code of clauses in subsequence order

  where the second and third levels are needed only depending on what sort of keys are present in the subsequence and in what number.

Optimizing the Design: Indexing

- Indexing code for subsequence *S1*:

```
          'first level indexing for subsequence S1'
          'second level indexing for subsequence S1'
          'third level indexing for subsequence S1'
     S11: try_me_else S12
          'code for call(or(X,Y)) :- call(X).'
     S12: retry_me_else S13
          'code for call(trace) :- trace.'
     S13: retry_me_else S14
          'code for call(or(X,Y)) :- call(Y ).'
     S14: retry_me_else S15
          'code for call(notrace) :- notrace.'
     S15: trust_me
          'code for call(nl) :- nl.'
```

Optimizing the Design: Indexing

- First level indexing makes control jump to a (possibly void) bucket of clauses, depending on whether *deref(A1)* is a:
  - **Variable:** the code bucket of a variable corresponds to full sequential search through the subsequence (thus, it is never void).
  - **Constant:** the code bucket of a constant corresponds to second level indexing among constants.
  - **Non-empty list:** the code bucket of a list corresponds to third level indexing among non-empty lists.
  - **Structure:** the code bucket of a structure corresponds to second level indexing among structures.

Optimizing the Design: Indexing Instructions

- First level indexing:
  - `switch_on_term V,C,L,S`

    jump to the instruction labeled *V*, *C*, *L*, or *S*, depending on whether *deref(A1)* is, respectively, a variable, a constant, a non-empty list, or a structure.
- Second level indexing (for N distinct symbols):
  - `switch_on_constant N,T`

    T is a hash-table of the form $\{c_i : L_i\}$, $1 \leq i \leq N$.

    If $deref\,(A1) = c_i$ then jump to instruction labeled L_i. Otherwise, backtrack.
  - `switch_on_structure N,T`

    T is a hash-table of the formform $\{s_i : L_i\}$, $1 \leq i \leq N$.

    if $deref\,(A1) = s_i$ then jump to instruction labeled L_i. Otherwise, backtrack.

Optimizing the Design: Indexing Instructions

- Third level indexing:
  - `try L`
  - `retry L`
  - `trust L`
- Identical to the `try_me_else/retry_me_else/trust_me` sequence, 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).

Optimizing the Design: Indexing Instructions

- Indexing code for subsequence *S1*:

```
      switch_on_term S11,C1,fail,F1       % first level indexing
  C1: switch_on_constant 3,               % second level indexing
             {trace: S1b,                 % for constants
              notrace: S1d,               %
              nl: S1e}                    %
  F1: switch_on_structure 1,              % second level indexing
             {or/2: F11}                  % for structures
 F11: try S1a                             % third level indexing
      trust S1c                           % for or/2
                                          %
 S11: try_me_else S12                     %
 S1a: get_structure or/2,A1               % A1 = or
      unify_variable A1                   %        (A1,
      unify_void 1                        %            _)
      execute call/1                      % call(A1)
```

Optimizing the Design: Indexing Instructions

- Indexing code for subsequence *S1*:

```
S12: retry_me_else S13              %
S1b: get_constant trace,A1          % A1 = trace
     execute trace/0                % trace
S13: retry_me_else S14              %
S1c: get_structure or/2,A1          % A1 = or
     unify_void 1                   %         (_,
     unify_variable A1              %            A1)
     execute call/1                 % call(A1)
S14: retry_me_else S15              %
S1d: get_constant notrace,A1        % A1 = notrace
     execute notrace/0              % notrace
S15: trust_me                       %
S1e: get_constant nl,A1             % A1 = nl
     execute nl/0                   % nl
```

Optimizing the Design: Indexing Instructions

- Indexing code for subsequence *S4*:

```
      switch_on_term S41,C4,fail,F4   % first level indexing
 C4:  switch_on_constant 2,           % second level indexing
          {repeat: C41,               % for constants
           true: S4d}                 %
 F4:  switch_on_structure 1,          % second level indexing
          {call/1: S4a}               % for structures
C41:  try S4b                         % third level indexing
      trust S4c                       % for repeat
```

Optimizing the Design: Indexing Instructions

- Indexing code for subsequence *S4*:

```
S41: try_me_else S42                    %
S4a: get_structure call/1,A1            % A1 = call
     unify_variable A1                  %           (A1)
     execute call/1                     % call(A1)
S42: retry_me_else S43                  %
S4b: get_constant repeat,A1             % A1 = repeat
     proceed                            %
S43: retry_me_else S44                  %
S4c: get_constant repeat,A1             % A1 = repeat
     put_constant repeat,A1             % A1 = repeat
     execute call/1                     % call(A1)
S44: trust_me                           %
S4d: get_constant true,A1               % A1 = true
     proceed                            %
```

Optimizing the Design: Summary

- New query instructions
  - `put_constant c,Ai`
  - `set_constant c`
  - `put_list Xi`
  - `put_list Ai`
  - `set_void N`

Optimizing the Design: Summary

- New program instructions
  - `get_constant c,Ai`
  - `unify_constant c`
  - `get_list Xi`
  - `get_list Ai`
  - `unify_void N`
  - `execute f/n`
  - `swith_on_term V,C,L,S`
  - `switch_on_constant N,T`
  - `switch_on_structure N,T`
  - `try L`
  - `retry L`
  - `trust L`