Implementation of Prolog

Implementation of Prolog

People	Location	Year	Type	Technique
Battani & Meloni	Marseille	1973	Fortran interpreter	structure sharing
David H.D. Warren	Edinburgh	1977	DEC-10 Prolog (native code)	structure sharing + multiple stacks + TRO
David H.D. Warren	SRI	1983	abstract machine + emulator	structure copying + goal stacking
David H.D.Warren	SRI	1984	WAM + emulator	structure copying + environment stacking

Implementation of Prolog

- Implementations based on the WAM: Quintus Prolog, Berkeley machine (PLM), NEC machine (HPM), ECRC machine (KCM).
- Prolog systems: SICStus Prolog, Arity Prolog, Mac Prolog, LPA Prolog, SWI Prolog, IC-Prolog, Turbo-Prolog, GNU-Prolog etc.

Implementation of Prolog

- Differences between compilation of Prolog programs and compilation of imperative languages:
  - logical variable (no destructive assignment. Once the variable is instantiated to a value, this can not change unless on backtracking).
  - backtracking (it does not recover space on procedure exit, unless it is executing the last clause of a predicate).
  - in imperative programming we remove the last execution stack on exit (call: push, exit: pop). In Prolog, stacks stay there till the last clause of a predicate is executed.

Implementation of Prolog

- WAM (Warren Abstract Machine)
  - Types of terms WAM: constant (integers or atoms), variable, structure, list, floating point.
  - Procedure: set of clauses with same name and arity.
  - Term Representation:

```
+---+---+
|TAG| N |
+---+---+
```

  - variable: REF can be a pointer to another variable, to a structure or to a list in the heap, or to itself (non-instantiated var)

```
+---+---+
|VAR|REF|
+---+---+
```

Implementation of Prolog

- constant: N, in general, is an index in a symbol table (normally implemented as a hash table).

```
+---+---+
|CTE| N |
+---+---+
```

- structure:

```
    +---+---+
    |STR| P |  P points to the structure
    +---+---+
    +---+---+
P:  |FN |ARI|  functor name is found in a table
    +---+---+  using index FN, ARI (arity of the term)
```

- list:

```
+---+---+
|LIS| P |
+---+---+
```

Implementation of Prolog

- Variable Classification:
  - regarding location during execution:
    - local: do not appear in functors (compound terms)
    - global: appear in functors

      `Ex: p(X,f(X,a),Y).   % X is global, Y is local`

Implementation of Prolog

- regarding lifetime:
  - temporary: appear only in the head and/or in the first literal of the clause body.
  - permanent: can appear in the head and after the first literal in the clause body.

```
Ex: d((U*V),X,((DU*V)+(U*DV))) :-
          d(U,X,DU),   % V, X and DV are permanent
          d(V,X,DV).   % U and DU are temporary
```

Implementation of Prolog

- regarding creation time:
  - conditional: created and not instantiated before a choicepoint. Can have different values depending on the alternative clauses in the choicepoint.
  - inconditional: is already instantiated when the choicepoint is created, therefore it does not change values.

Implementation of Prolog

- Components of the abstract machine:
  - Instruction set
  - Registers
  - Memory areas:
    - code + data
    - local stack: stores information about environments, choicepoints and local variables
    - Heap (global): stores structures (compound terms) and variables that appear in structures (global variables)
    - Trail: stores addresses of conditional variables: those that need to be unbound upon failure of a clause of a predicate
- algorithms: dereferencing, unification and backtracking

Implementation of Prolog

- Structure sharing: structures are not copied to the heap. Instead, the variables are copied. There are pointers from the runtime environment to the code area.
- Struture copying: there is no pointer from the execution environment to the code area.
- Structure Sharing x Structure Copying:
  - Sharing saves memory, but it can lose locality
  - Copying uses more space, but it can win in locality

Implementation of Prolog

- Registers:

Reg	purpose	pointer to
P	program counter	code area
CP	continuation pointer	code area
E	local stack top	environment stacking
B	last choicepoint	local stack
TR	trail top	trail
H	heap top	heap
HB	last choicepoint	heap
S	heap structure being unified	heap
Xi	arguments of second and upper levels	code area
Ai	arguments of first level	code area
Yi	local variables	code area

Implementation of Prolog

- Instruction Set:

Control:

allocate	allocates space to local variables
call P/n, N	prepares environment with label P/n indicating that there are N local variables in the stack
execute P/n	jumps to label P/n

First level unification:

get_variable Ri,Aj	creates a slot for a variable in the heap (if Ri is Xi) or in the stack (if Ri is Yi), dereference whatever comes in Aj and unifies with Ri get_value Ri,Aj dereferences Ri and Aj and unifies Ri with Aj
get_constant c,Ai	dereferences Ai and unifies with c
get_structure F/n,Ai	dereferences Ai and unifies with F/n

Implementation of Prolog

- Instruction Set:

  Unification of upper level terms:

unify_variable Ri	dereferences what is pointed by S, creates slot in the heap, Ri points to that slot, unifies S with Ri
unify_constant c	dereferences whatever is pointed by S, unifies with c
unify_structure F/n	dereferences whatever is pointed by S, unifies with F/n and invokes unification again

Implementation of Prolog

- Instruction Set:
  PUT Instructions: transfer values from arguments to registers
  put_variable Ri,Aj
  put_constant c,Aj
  put_structure F/n,Aj

Implementation of Prolog

- Instructions *_variable are generated for the first occurrence of a variable in the clause. The subsequent occurrences of the same variable generate *_value.
- Instructions *_constant are generated when constants are found in the code.
- Instructions *_structure are generated when we find compound terms of first or upper level in the code.
- Instructions for choicepoints: generated at each clause entry and only executed if the argument that comes is an unbound variable (in this case, can not index and jump directly to a given clause)

  try_me_else L, retry_me_else L, trust_me

Implementation of Prolog

- Indexing instructions:

```
first level:
switch_on_term Lv,Lc,Ll,Ls
dereferences register A1 (first argument). If it is:
-- variable, jump to Lv
-- constant, jump to Lc
-- list, jump to Ll
-- structure, jump to Ls

second level:
switch_on_constant N, {c1:L1, c2:L2,...,cn:LN}
switch_on_structure N, {s1:L1, s2:L2,...,sn:LN}

third level:
try L
retry L
trust L
```

Implementation of Prolog

- Indexing instructions are generated by the compiler after the code for all clauses of a predicate is generated.
- Although these instructions index only on the first argument, there are Prolog implementations that generate code for indexing on more than the first argument.

Implementation of Prolog

- Read and Write Mode in the unification instructions:
  - Read Mode: used for unification of already existing structure.
  - Write Mode: to build a new structure.

```
In read mode:
  unify_variable X    X := next argument of S
  unify_value X       unify X with next argument of S
  unify_constant C    unify C with next argument of S

In write mode:
  unify_variable X    X := reference to next arg of H :=
  unify_value X       next argument of H := X
  unify_constant C    next argument of H := C
```

Implementation of Prolog

- General form of a Prolog compiled code:

```
p :- q, r, s.

allocate
get args de P
put args de q
call q, n
put args de r
call r, n1
put args de s
deallocate
execute s
```

```
p :- q

get args de p
put args de q
execute q

p.

get args de P
proceed
```

Implementation of Prolog

```
gf(X,Z) :- parent(X,Y), parent(Y,Z).
parent(joao,maria).
parent(joao,jose).
parent(jose,maria).
?- gf(joao,X).

consulta/0:     allocate                % query code
                put_constant joao,A1    % gf(joao,
                put_variable Y1,A2      % X
                call gf/2,1             %
                proceed                 % ).

parent/2:
parent/2_1:     try_me_else parent/2_2          % code for first fact
                get_constant joao, A1
                get_constant maria, A2
                proceed
parent/2_2:     retry_me_else parent/2_3        % code for second fact
                get_constant joao, A1
                get_constant jose, A2
                proceed
parent/2_3:     trust_me                        % code for third fact
                get_constant jose, A1
                get_constant maria, A2
                proceed
gf/2:           allocate                        % code for rule gf/2
                get_variable Y1,A1
                get_variable Y2,A2
                put_value Y1,A1
                put_variable Y3,A2
                call parent/2,2
                put_value Y3,A1
                put_value Y2,A2
                call parent/2,2
                proceed
```

Implementation of Prolog

```
Structures:
----------

?-p(Z,h(Z,W),f(W)). % first level : A1 = Z, A2 = h(Z,W), A3 = f(W),
                    % second level: X4 = Z, X5 = W

p(f(X),h(Y,f(a)),Y). % first level : A1 = f(X), A2 = h(Y,f(a)), A3 = Y
                     % second level: X4 = X, X5 = Y, X6 = f(a)
                     % third level : X7 = a

consulta/0:     put_variable X4,A1      % p(Z, Z was renamed to X4
                put_structure h/2,A2    %   h
                set_value X4            %    (Z,
                set_variable X5         %     W),
                put_structure f/1,A3    %   f(
                set_value X5            %     W
                call p/3,0              %    ).

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

Implementation of Prolog

```
List concatenation:
------------------

app([],L,L).
app([X|L1],L2,[X|L3]) :- app(L1,L2,L3).


app/3:
        switch_on_term C1a,C1,C2,fail

C1a:    try_me_else C2a         % app(
C1:     get_constant nil,A1     % [],
        get_value A2,A3         % L, L
        proceed                 % ).

C2a:    trust_me                % app(
C2:     get_list A1             % [
        unify_variable X4       %  X|
        unify_variable A1       %  L1], L2,
        get_list A3             % [
        unify_value X4          %  X|
        unify_variable A3       %  L3]) :-
        execute app/3           % app(L1,L2,L3).
```

NB: This code is extremely optimized. Various instructions that would normally be generated do not appear in this code.
Exercise: generate the normal code for app/3 without optimizations.

Implementation of Prolog

```
Quicksort:
---------

qs([],R,R).
qs([X|L],R0,R) :-
    split(L,X,L1,L2),
    qs(L1,R0,[X|R1]),
    qs(L2,R1,R).

qs/3:     switch_on_term C1a,C1,C2,fail

C1a:      try_me_else C2a       % qs(
C1:       get_constant nil,A1   % [],
          get_value A2,A3       % R, R
          proceed               % ).

C2a:      trust_me              % qs(
C2:       allocate
          get_list A1           % [
          unify_variable Y6     %   X|
          unify_variable A1     %   L],
          get_variable Y5,A2    % R0,
          get_variable Y3,A3    % R) :-
          put_value Y6,A2       % split(L,X,
          put_variable Y4,A3    %           L1,
          put_variable Y1,A4    %           L2
          call split/4,6        % ),
          put_unsafe_value Y4,A1 % qs(L1,
          put_value Y5,A2            R0,
          put_list A3           % [
          unify_value Y6        %   X|
          unify_variable Y2     %   R1]
          call qs/3,3       % ),
          put_unsafe_value Y1,A1 % qs(L2,
          put_value Y2,A2       % R1,
          put_value Y3,A3       % R
          deallocate
          execute qs/3          % ).
```

```
NB: Again, this code is extremely optimized, including
    reuse of registers to minimize their use. The instruction
    put_unsafe_value is used to save variables that in the stack
    in the heap, in case there is reutilization of space (instruction
    execute may reutilize the current stack environment).
```