Υλοποίηση Γλωσσών Λογικού Προγραμματισμού

Κωστής Σαγώνας <kostis@cs.ntua.gr>

Περιεχόμενα

- Ιστορία των υλοποιήσεων της Prolog
- Η εικονική μηχανή του Warren (WAM)
  - Αναπαράσταση όρων και μεταβλητών
  - Πέρασμα παραμέτων
  - Εντολές
  - Υλοποίηση της οπισθοδρόμησης

History of Prolog Implementations

- Prolog was first implemented in the early 70's as an interpreter based on automated theorem proving technology
- The first compiler, the DEC-10 Prolog, was implemented by David H. D. Warren in 1977
- In the early 80's, came the Warren Abstract Machine or WAM
- Since then, most Prolog implementations have been WAM-based
  - the fastest (Aquarius, Parma) used their own variations of the WAM
- Most WAM-based Prologs have been bytecode interpreters (e.g., SWI-Prolog, SICStus, XSB , YAP), although some have compiled the WAM to native code (e.g., Prolog_by_BIM, SICStus, GNU Prolog)

The WAM: Warren Abstract Machine

- The original report on the WAM (1983) described how the WAM worked, but not why
  - Probably less than 50 people in the whole world understand it!
- The WAM is ingenious but quite complex; some of its components depend for correctness on details of the Prolog language definition (e.g., the computation rule)
- The WAM is hard to modify or further optimize
- These days, most Prolog implementations come with extensions of some kind:
  - coroutining, support for cyclic terms, multi-threading, constraints, tabling, etc.

WAM Data Areas

- The **heap** stores compound terms (lists and structures)
- The **stack** contains an irregular interleaving of

  Environments: an environment contains the state of a clause, the values of the local variables, and the return address.

  Choice points: a CP records the state of the abstract machine (including all the argument registers) when calling a predicate with more than one matching clauses. It also points to the next matching clause.
- The **trail** records which variables need to be reset to unbound on backtracking

Term Representation

- Terms are represented by tagged words
- The tag is typically 3 to 6 bits in size and often stored on the least significant bits of the word

  INT: the rest of the word is an integer

  FLOAT: the rest of the word is a floating point number

  ATOM: the rest points to an entry in the **atom table** where the name of the atom is stored

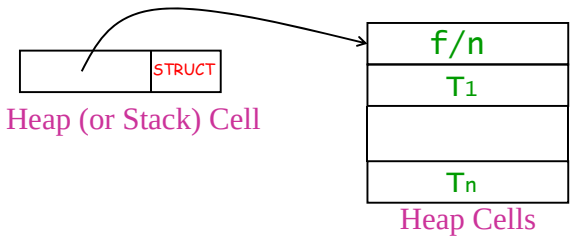
  LIST: the untagged word is a pointer to a heap cons cell

  STRUCT: the untagged word is a pointer to a heap cell where a compound term (other than a list) is stored

  VAR: the word represents a variable

Term Representation: Compound Terms

`f(T1,…,Tn)` is represented as a structure pointing to a heap location containing `n+1` cells. The first specifies the function symbol `f/n`, the others contain the arguments.



Due to tags, it is always possible to tell what term a value stored in a word represents *without* keeping any extra type information.

Term Representation: Variables in the WAM

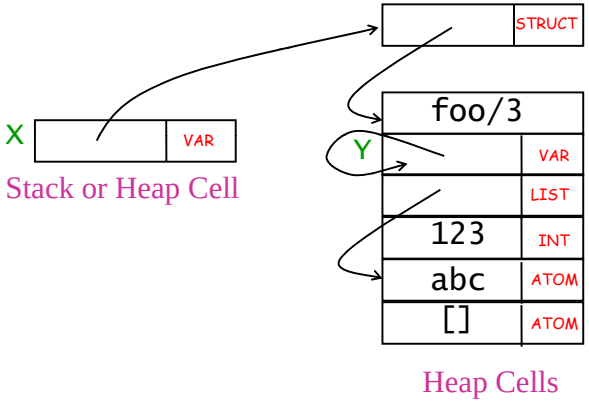
- An unbound variable points to itself
- A variable bound to a term (which may be another variable) points to that term
- Unifying two variables leaves the one unchanged and makes the other one point to it

Rules about pointer directions prevent dangling pointers

1. Heap to heap references and stack to stack references must all point from younger to older data
2. Stack cells can point to the heap but not vice versa

Term Representation: An Example

```
X = foo(Y,[abc],123)
```



A = B in the WAM

Suppose both A and B are unbound before executing

A = B

Dereferencing

- Whenever a WAM operation operates on a variable, it must start by *dereferencing* it:
  - Follow the chain of variable-to-variable bindings to its end (until either a non-variable term or an unbound variable is found)
- Even though most chains are very short (containing zero or one variable-to-variable bindings), there is no bound on the chain length, so the dereference code must contain a loop

Term Representation: Variables in PARMA

To avoid the cost of dereferencing, a different representation for variables can be used (PARMA scheme)

- An unbound variable still points to itself, but unifying two variables creates a circular chain of references, not a linear one
- When unifying a variable with a non-variable term, all variables in the chain are made to point to the non-variable term. Thereafter, one can lookup the values without dereferencing

Since the values of many variables are read only once and some variables never have their values read, in practice the speedup by adopting the PARMA scheme is smaller than one would expect

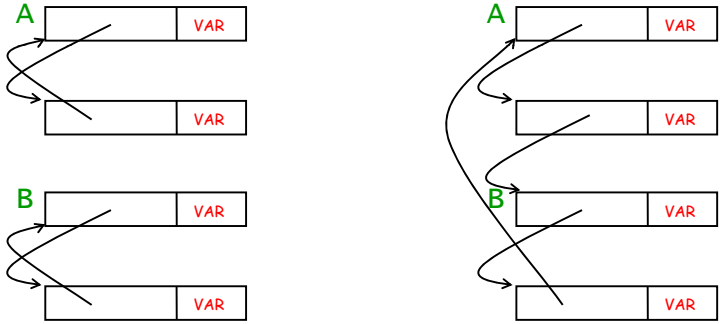
A = B in PARMA

Suppose both A and B are unbound before executing

A = B

A = B in PARMA (cont.)

Suppose A and B are each bound to other variables before executing A=B



Parameter Passing (caller)

The WAM has an array of abstract machine registers $A_1,\ldots,A_n$.
When calling a predicate, the caller puts the i-th argument in register A_i. Any of the A_i's that are not needed to hold arguments can be used as temporaries.

To pass the term `foo(Y,abc,123,Y)` as the first argument to `p/3`:

```
put_structure A1, foo/4
set_variable  A4
set_constant  abc
set_constant  123
set_value     A4
```

The code uses A_4 as a temporary holding the value of `Y`

Constructing Terms

- The put_structure instruction allocates only the cell header, the word pointing to the cell and the word identifying the function symbol. The arguments have to be filled in by the following instructions, each of which allocates one word on the heap.
- The set_variable instruction creates an unbound variable on the heap, and makes A_4 point to it. The set_value instruction copies the value in A_4 to the heap.
- More complex terms can be built from the inside out.
- For each term the sequence is always the same: place the function symbol, and then each of its arguments created with set_variable, set_value, and set_constant.

Parameter Passing (callee)

The code to unify the term in a given argument position in the head with the value passed by the caller has the same structure as the code to create the term:

```
p(tree(X,L,R),Y) :- …
```

```
get_structure   A1, tree/3
unify_variable  A3
unify_variable  A4
unify_variable  A5
…
```

However, this code must work whether the caller calls `p(A,B)` where `A` is an unbound variable, or `p(tree(1,nil,nil),3.14)`.

Read vs. Write Mode

- The get_structure instruction starts by dereferencing A_1 and checking whether it is free
  - If it is free, it sets the current mode to WRITE. This makes the rest of the get_structure behave like put_structure, and it makes the subsequent unify_variable instructions behave like set_variable.
  - If it is bound, it sets the current mode to READ. This makes the rest of the get_structure and the subsequent unify_variable instructions do matching against the existing term, instead of constructing a new one.
- More complex terms can be matched from the outside in (i.e., outermost term first).

General Unification

```
p(f(X,X)) :- …
```

```
get_structure A1, f/2
unify_variable  A2
unify_value  A2
…
```

- When `p/1` is passed a ground term as its argument, the unify_variable instruction picks up the value of the first argument of `f/2`. The unify_value instruction invokes the general unification routine to unify this value with the second argument of `f/2`.
- The unification routine usually omits the occur check. In some cases, circular terms can be created that the rest of the WAM cannot handle.

Clause Code Structure

1. An allocate instruction which allocates an environment. Its argument specifies how many stack slots to reserve for variables (**Y** registers) that need to be saved across calls.
2. A sequence of get_* and unify_* instructions to match the arguments of the clause head.
3. A sequence of calls in the body. Each starts with put_* and set_* instructions to set up the arguments and ends with a call instruction.
4. A deallocate instruction which deallocates the environment and branches to the return address saved in it by the allocate.

NOTE: get_* and put_* instructions specify a register or stack slot as an operand. In unify_* and set_* instructions, this operand is implicitly the heap cell which is after the heap cell involved in the last get_* and put_* instruction.

Compiling Clauses: Example 1

```
p(U,abc,f(V,U)) :- s, q(g(V)), r(U).
```

```
allocate          2
get_variable      A1, Y1
get_constant      A2, abc
get_structure     A3, f/2
unify_variable    Y2
unify_value       Y1
call              s/0
put_structure     A1, g/1
set_value         Y2
call              q/1
put_value         A1, Y1
call              r/1
deallocate
```

The first references to each of U and V use a *_variable instruction, while later references use a *_value instruction.

Environments and Variable Classification

- Clauses need an environment (stack frame) only if they contain calls to more than one predicate in their body.
- The stack frame contains slots for variables whose values need to be preserved across calls.
- These variables are called permanent.
- All other variables are temporary.
- Temporary variables are placed in argument registers.

Compiling Clauses: Example 2

```
p(U,abc,f(V,U)) :- q(g(V)), r(U).
```

```
allocate          1
get_variable      A1, Y1
get_constant      A2, abc
get_structure     A3, f/2
unify_variable    A4
unify_value       Y1
put_structure     A1, g/1
set_value         A4
call              q/1
put_value         A1, Y1
call              r/1
deallocate
```

Only variable U is permanent here!

Compiling Multiple Clauses

- If a predicate has more than one clause, we can compile each clause separately and link the code together with a try/retry/trust chain: try for the first clause, trust for the last, and retry for all the others in the middle.
- The try_me_else instruction creates a choice point.
  - This prepares for backtracking to the next clause by saving all the abstract machine registers whose values need to be reset upon backtracking; e.g., the top of heap and top of trail registers, and the A_i registers containing the arguments.
- The retry_me_else instruction updates the next alternative field of the existing choice point.
- The trust_me instruction deletes the existing choice point.

Compiling Multiple Clauses: Example 1

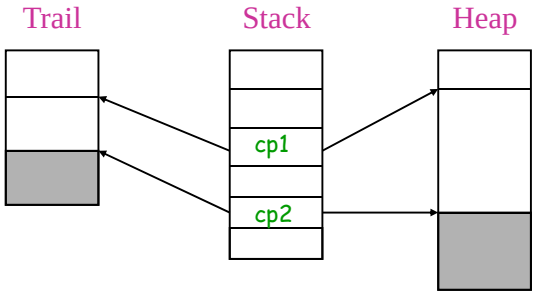
```
p/3
      try_me_else  p3clause2
      <code for clause 1>
p3clause2
      retry_me_else     p3clause3
      <code for clause 2>
p3clause3
      retry_me_else     p3clause4
      <code for clause 3>
p3clause4
      trust_me
      <code for clause 4>
```


Compiling Multiple Clauses: Example 2

```
p/3
      try       p3clause1
      retry     p3clause2
      retry     p3clause3
      trust     p3clause4

p3clause1
      <code for clause 1>
p3clause2
      <code for clause 2>
p3clause3
      <code for clause 3>
p3clause4
      <code for clause 4>
```


Implementation of Backtracking



- When the program backtracks to e.g. cp2, it will throw away the part of the heap that was allocated since cp2 was created.
- However, the code executing since the creation of cp2 may also have modified the retained part of the heap by binding some of the variables stored there.

The Trail

- When a WAM operation binds a variable, and the variable is older than the most recent choice point (i.e., occurs in the retained part of the stack), we record its address on the trail.
- When backtracking to a CP, we also *unwind* the segment of the trail that was created since the creation of the CP.
  - Unwinding a trail entry means resetting the variable pointed to by the trail entry to unbound, and then disregarding the trail entry.
- The PARMA variable representation requires *value trailing*
  - a trail entry must contain not just the address of the bound variable, but also its old contents;
  - unwinding must restore the old value.

Memory Management: Instant Reclamation

Backtracking recovers some memory from the heap, from the stack, and from the trail.

- This is done reasonably efficiently.
- The rules about pointer direction in variables exist to ensure that a variable whose storage is recovered by backtracking is never pointed to by another variable whose storage remains allocated.

Memory Management: Garbage Collection

- The main task of garbage collection is to recover from the heap memory cells that won't be needed again, even after backtracking.
- Some collectors also collect trail entries referring to unreachable variables, and some collect unreachable environments and choice points.
- The most thorough also collect entries from the atom table (atom garbage collection).

Protecting Environments

```
p(A,D) :- q(A,B), r(A,C), s(B,C,D).
```

- If `r/2` leaves a choice point, then we cannot disregard `p`'s environment when `p/2` returns. If we did, we would not know what the value of `B` should be in the call to `s/3` after `r/2` possibly succeeds for the second time.
- The WAM's **deallocate** instruction therefore discards an environment only if there are no choice points on top of it.
- The instructions that allocate environments and choice points always do so above both the topmost environment and the topmost choice point.

Last Call Optimization

- Most Prolog clauses end with a predicate call.
- In the usual case, the code for the clause will deallocate the clause's environment after it has set up the arguments for the last call. It will also arrange for the called predicate to return not to its caller but to the caller's caller.
- Tail recursion optimization (TRO) is a special case of last call optimization (LCO).
- Unfortunately, if one of the calls in the body of the clause has left a choice point, TRO and LCO are both disabled.

Compiling Clauses with Last Call Optimization

```
p(U,abc,f(V,U)) :- q(g(V)), r(U).
```

allocate	1
get_variable	Y_1
get_constant	A_2, abc
get_structure	A_3, f/2
unify_variable	A_4
unify_value	Y_1
put_structure	A_1, g/1
set_value	A_4
call	q/1
put_value	A_1, Y_1
call	r/1
deallocate	

allocate	1
get_variable	A_1, Y_1
get_constant	A_2, abc
get_structure	A_3, f/2
unify_variable	A_4
unify_value	Y_1
put_structure	A_1, g/1
set_value	A_4
call	q/1
put_unsafe_value	A_1, Y_1
deallocate	
execute	r/1

Variable U, which is permanent, needs to be moved!