

Logic Programming Implementation Part I: The WAM

Kostis Sagonas
kostis@it.uu.se

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 based on the WAM although 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**), although some have compiled the WAM to native code (e.g., **Prolog_by_BIM**, **SICStus**, **GNU Prolog**)

Implementation of Logic Programming Languages

2

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.

Implementation of Logic Programming Languages

3

Forms of Interpreters

- Source level interpreters are very slow because they spend much of their time in doing lexical analysis
- Interpreters that operate on parse trees avoid this cost, but they still have to do much list scanning
 - e.g. when implementing a 'goto' or 'call'
- By compiling the program to the instruction set of an abstract machine and adding a table that maps names and labels to addresses in this program, some of the interpretation overhead can be reduced
- For convenience, most abstract instruction sets use integral numbers of bytes to represent everything
 - opcodes, register numbers, stack slot numbers, indices into the predicate or atom table, etc.

Implementation of Logic Programming Languages

4

Structure of Bytecode Interpreters

```
byte *pc = &byte_program[0];
while(1) {
  opcode = pc[0];
  switch (opcode) {
    ...
    case GET_CONSTANT:
      source_reg_num = pc[1];
      atom_num_to_match = get_2bytes(&pc[2]);
      ...
      pc += 4;
      break;
    ...
    case JUMP:
      jump_addr = get_4bytes(&pc[1]);
      pc = &byte_program[jump_addr];
      break;
    ...
  }
}
```

Implementation of Logic Programming Languages

5

WAM Data Areas

- The **heap** stores compound terms
- 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, and points to the next matching clause
- The **trail** records which variables need to be reset to unbound on backtracking

Implementation of Logic Programming Languages

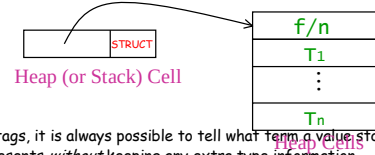
6

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

$f(T_1, \dots, T_n)$ 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

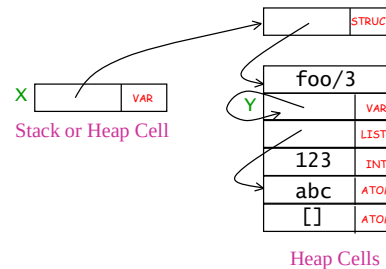
- An unbound variable points to itself
- A variable bound to a term (which may be another variable) points to that term
- Unifying two variables leaves the one unchanged and makes the other one point to it

Rules about pointer directions prevent dangling pointers

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

Term Representation: An Example

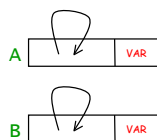
$x = \text{foo}(Y, [\text{abc}], 123)$



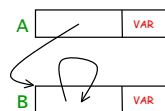
A = B in the WAM

Suppose both A and B are unbound before executing

$A = B$



after 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
- This is *distributed fat*: a cost that all parts of the program must incur, even though only a few parts benefit from it

Term Representation: Variables in PARMA

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

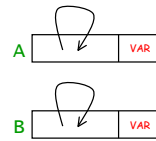
- An unbound variable still points to itself, but unifying two variables creates a circular chain of references, not a linear one
- When unifying a variable with a non-variable term, all variables in the chain are made to point to the non-variable term. Thereafter, one can lookup the values without dereferencing

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

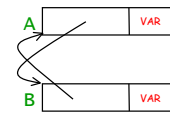
A = B in PARMA

Suppose both A and B are unbound before executing

A = B

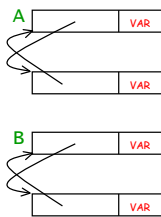


after executing A = B

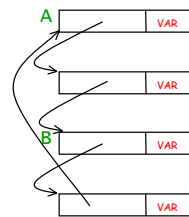


A = B in PARMA (cont.)

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



after executing A = B



Parameter Passing (caller)

The WAM has an array of abstract machine registers $A_1, \dots, A_n$.

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)) :- ...`

```
get_structure A1, tree/3
unify_variable A2
unify_variable A3
unify_variable A4
...
```

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

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

- 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.
- A sequence of `get_*` and `unify_*` instructions to match the arguments of the clause head.
- 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.
- 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 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 `Ai` registers containing the arguments.
- The `retry_me_else` instruction updates the next alternative field of the existing choice point.
- The `trust_me_else` instruction deletes the existing choice point.

Compiling Multiple Clauses: Example 1

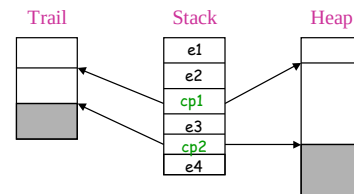
```
p/3:
  try_me_else  p3clause2
  <code for clause 1>
p3clause2:
  retry_me_else p3clause3
  <code for clause 2>
p3clause3:
  retry_me_else p3clause4
  <code for clause 3>
p3clause4:
  trust_me
  <code for clause 4>
```

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 in calls.
- 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).
```

Without LCO		With LCO	
allocate	1	allocate	1
get_variable	A1, Y1	get_variable	A1, Y1
get_constant	A2, abc	get_constant	A2, abc
get_structure	A3, f/2	get_structure	A3, f/2
unify_variable	A4	unify_variable	A4
unify_value	Y1	unify_value	Y1
put_structure	A1, g/1	put_structure	A1, g/1
set_value	A4	set_value	A4
call	q/1	call	q/1
put_value	A1, Y1	put_unsafe_value	A1, Y1
call	r/1	deallocate	
deallocate		execute	r/1

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

Choice Points

Unnecessary choice points are a performance problem mainly because they prevent the reclamation of both heap and stack space.

There are two ways to deal with choice points:

- Remove them, by backtracking or by using a pruning operator such as if-then-else or cut
- Avoid creating them in the first place

The latter is obviously preferable.

Efficiency Problem: Unnecessary CP Creation

- Choice points are created when calling a predicate with more than one clause.
- They are removed when backtracking enters the last clause, or when the program calls cut (!/0).

```
max(A,B,A) :- A > B.
max(A,B,B) :- A <= B.

?- max(2,1,X). % CP created and left
?- max(1,2,X). % CP created and removed
```

The second query does not leave a CP. However, it does cause a CP to be created and filled in. Since CPs are large, this takes some time.

Typical "Solution": Cutting away the CP

```
max(A,B,A) :- A > B, !.  
max(A,B,B).
```

- This code is hard to read, because each clause cannot be read independently (on its own).
 - The second clause is not consistent with the intended semantics
- It is also not *steadfast*: it does not work if the third argument is actually input.

```
?- max(2,1,X).  
X = 2;  
no  
?- max(2,1,1).  
yes % succeeds!
```

Implementation of Logic Programming Languages

37

A Solution: Making Code Steadfast

- A predicate definition is *steadfast* if its success set (the set of atoms for which it succeeds) does not change by supplying as input arguments that are intended to be output.
- Code can be made steadfast by delaying unifications with output arguments until after the cut.

```
max(A,B,M) :- A > B, !, M = A.  
max(A,B,B).  
  
?- max(2,1,X).  
X = 2;  
no  
?- max(2,1,1).  
no % fails as it should
```

Implementation of Logic Programming Languages

38

A Better Solution: If-Then-Else

```
max(A,B,M) :-  
    ( A > B -> M = A ; M = B ).
```

- If-then-else is defined in terms of cut, so this code has the same behavior as the code on the previous slide
- However, using if-then-else naturally leads to code that is more readable, more logical, and also steadfast. Several systems avoid creating CPs for such if-then-elses, so the code can also be faster.

Implementation of Logic Programming Languages

39

Efficiency Problem: Unnecessary CP Creations

```
%% ith(+Index, +List, ?Element).  
  
ith(1,[H|_],H).  
ith(I,[_|T],E) :-  
    I > 1, I1 is I-1, ith(I1,T,E).
```

- Most Prolog systems cannot recognize that *ith/3* is deterministic and typically create a choice point for this code.
- Placing a cut in the first clause is a partial solution to this problem as the CP is first created and then cut away.

Implementation of Logic Programming Languages

40

Avoiding Choice Point Creation with If-Then-Else

- In contrast, many Prolog systems will not create a choice point for uses of if-then-else where the conditions are Prolog built-ins that do not create bindings of variables.

```
%% ith(+Index, +List, ?Element).  
  
ith(I,[H|T],E) :-  
    ( I =:= 1 -> E = H  
    ; I > 1, I1 is I-1, ith(I1,T,E)  
    ).
```

Implementation of Logic Programming Languages

41

Efficiency Problem: Unnecessary CP Creations

```
ordered([A,B|T]) :- A <= B, ordered([B|T]).  
ordered([_]).  
ordered([]).
```

This is a *very* inefficient predicate to check whether a list of numbers is ordered.

1. Every call (except the base cases) leaves a CP on the stack
2. Every recursive call uses stack space
3. Reclaiming of heap and stack space will be disabled in predicates which call *ordered/1*.

Implementation of Logic Programming Languages

42

Avoiding Choice Points by Rewriting the Code

- Swapping the clauses of `ordered/1` results in only the call for the 1-element list to leave a CP, but that is still one CP too many!
- Clause indexing can be used to avoid all choice points in `ordered/1` if it is rewritten in the form suggested by induction on the structure of the list type.

```
ordered([]).
ordered([_|T]) :- ordered_cons(T,H).
% clause indexing (on 1st argument) => no CP
ordered_cons([],_).
ordered_cons([B|T],A) :- A <= B, ordered_cons(T,B).
```

Implementation of Logic Programming Languages

43

Clause Indexing

- Clause indexing breaks the clauses into different subsets which could match with different clauses or calls
 - If the first argument to `ordered_cons/2` was a constant, only the first clause could possibly match
 - If the first argument to `ordered_cons/2` was a compound term, only the second clause could possibly match
 - If the first argument to `ordered_cons/2` was a variable, both clauses would match
- Most Prolog systems index on the top-level functor of the first argument of a predicate
 - So, if `ordered_cons/2` is called with a non-variable, no CP is created

Implementation of Logic Programming Languages

44

Clause Indexing Instructions of the WAM

- The WAM has several instructions for indexing which are typically used at the start of the code for a predicate
 - The `switch_on_term` instruction's arguments are a register and four labels. It inspects the tag of the register and jumps to one of the labels, depending on whether the register contains a variable, a list, or another structure.
 - The `switch_on_constant` instruction's arguments are a register, a hash table mapping constants to labels, and a default label. It looks up the constant in the register and jumps to the corresponding label. If the constant is not found, it goes to the default label (`fail`).
- Many implementations only index on the top-level functor of the first argument; in these the register is implicitly `A1`.

Implementation of Logic Programming Languages

45

Clause Indexing: An Example

```
p(c,X) :- ...
p(f(X),Y) :- ...
p(g(X,Z),Z) :- ...
```

```
p/2:
    switch_on_term A1, p2var, p2clause1, fail, p2struct
p2var:
    try      p2clause1
    retry    p2clause2
    trust    p2clause3
p2struct:
    switch_on_constant A1, {f/1->p2clause2, g/2->p2clause3}, fail
p2clause1: <code for clause 1>
p2clause2: <code for clause 2>
p2clause3: <code for clause 3>
```

Implementation of Logic Programming Languages

46

Indexing and Variables

```
p(f(X),...) :- ...
p(X,...) :- ...
p(g(X,Z),...) :- ...
```

```
p/2:
    switch_on_term A1, p2var, p2clause2, p2clause2, p2struct
p2var:
    try      p2clause1
    retry    p2clause2
    trust    p2clause3
p2struct:
    switch_on_constant A1, {f/1->p2f1, g/2->p2g2}, p2clause2
p2f1:
    try      p2clause1
    trust    p2clause2
p2g2:
    try      p2clause2
    trust    p2clause3
```

Implementation of Logic Programming Languages

47

Processing a Defaulty Data Structure

```
p(foo1(...),...) :- ...
p(foo2(...),...) :- ...
...
p(fooN(...),...) :- ...
p(X,...) :- ...
```

- The last clause typically applies only if no other clause matches
- Using `N` inequalities in the last clause is tedious, hard to maintain and leaves a choice point
- If-then-else leads to a large, hard to read clause, and maybe lots of CP creation and removal
- Tempting to use cuts in the first `N` clauses
 - but watch out for non-steadfast code

Implementation of Logic Programming Languages

48

Defaulty Data Structures

The best solution is usually to redesign the data structure

- $N+k$ cases should lead to $N+k$ functors, not N functors and a default(y) case
 - Leads to fewer bugs and indexing avoids the creation of CPs
- We are however sometimes stuck with a defaulty data structure that we cannot change
 - such as Prolog goals, Prolog arithmetic expressions, or the terms returned by numbervars
- Sometimes, a defaulty data structure yields efficiency benefits
 - A tree with variables as its leaves can be updated in place by instantiating a leaf
 - Instead of paying in updates, this scheme pays on lookups

Implementation of Logic Programming Languages

49

Restructuring Defaulty Code

If we can test for the default case (e.g., in Prolog arithmetic expressions, they represent numbers), the following structure is preferable:

```
p(X,Y) :-  
    ( default(X) ->  
      ...  
      ;    p1(X,Y)  
    ).  
  
p1(foo1(...),...) :- ...  
p1(foo2(...),...) :- ...  
...  
p1(fooN(...),...) :- ...
```

Implementation of Logic Programming Languages

50

Restructuring Defaulty Code (cont.)

If we can not test for the default case:

```
p(X,Y) :-  
    ( p1(X,Yprime) ->  
      Y = Yprime  
      ;    ...  
    ).  
  
p1(foo1(...),...) :- ...  
p1(foo2(...),...) :- ...  
...  
p1(fooN(...),...) :- ...
```

Implementation of Logic Programming Languages

51

Saving Heap Space

The heap grows when

1. A variable in a call is unified with a compound term in a clause head
 2. A call syntactically containing a compound term is executed
 3. Some built-in predicates that create terms are called
- The heap shrinks on backtracking or by garbage collection

Implementation of Logic Programming Languages

52

Saving Heap Space (cont.)

Things to avoid:

1. Algorithms that create intermediate data structures (integers and atoms are OK)
2. Wrapping terms in a compound term, especially if the terms are being "modified"
3. `=./2` ("univ")

Implementation of Logic Programming Languages

53

Saving Heap Space (cont.)

Avoiding wrappers:

```
p(name(S,G,M)) :-  
    ...  
    q(name(S,G,M)), % uses heap space
```

Most Prolog systems do not keep track of which terms have occurred in the clause so far, so the call will create a new term on the heap

```
p(S,G,M) :-  
    ...  
    q(S,G,M), % does not use heap
```

Unfortunately, this more efficient version is less abstract (the representation of the name is exposed) and harder to modify

Implementation of Logic Programming Languages

54

Saving Heap Space (cont.)

A compromise:

```
p(Name) :-  
    name(Name,S,G,M),  
    ...  
    q(Name),    % does not use heap  
% no heap used if first argument is input  
name(name(S,G,M),S,G,M).
```

One can also use an explicit unification such as

```
    Name = name(S,G,M),  
in the body for this purpose.
```

Assert/Retract

- Asserting or retracting a clause in a compiled predicate definition generally requires recompiling the predicate.
- Most Prolog systems allow a predicate's definition to be changed with assert/retract only if the predicate is declared dynamic
 - Some interpret dynamic predicates instead of compiling them.
- One cannot give a good, simple semantics to programs that modify the definition of a predicate that is currently executing. If a **trusted** clause adds a new clause, it will not affect the current call. If a non-**trusted** clause adds a new clause, it might affect the current call.
 - Researchers have devised *consistent* semantics that avoid such anomalies, but few Prolog systems implement them.