Knowledge Representation and Reasoning
Logics for Artificial Intelligence

Stuart C. Shapiro

Department of Computer Science and Engineering
and Center for Cognitive Science
University at Buffalo, The State University of New York
Buffalo, NY 14260-2000

shapiro@cse.buffalo.edu

Contents

6 Prolog

Horn Clauses

A Horn Clause is a clause with at most one positive literal.

Either $\{\neg Q_1(\overline{x}), \ldots, \neg Q_n(\overline{x})\}$ (negative Horn clause)

or $\{C(\overline{x})\}$ (fact or positive or definite Horn clause)

or $\{\neg A_1(\overline{x}), \ldots, \neg A_n(\overline{x}), C(\overline{x})\}$ (positive or definite Horn clause)

which is the same as

$A_1(\overline{x}) \wedge \cdots \wedge A_n(\overline{x}) \Rightarrow C(\overline{x})$

where $A_i(\overline{x})$, $C(\overline{x})$, and $Q(\overline{x})$ are atoms.

SLD Resolution

Selected literals, Linear pattern, over Definite clauses

SLD derivation of clause c from set of clauses S is
$c_1, \ldots, c_n = c$
s.t. $c_1 \in S$
and c_{i+1} is resolvent of c_i and a clause in S. [B&L, p. 87]

If S is a set of Horn clauses,
then there is a resolution derivation of $\{\}$ from S
iff there is an SLD derivation of $\{\}$ from S.

SLDSolve

```
procedure SLDSolve(KB,query) returns true or false {
   /* KB = {rule_1, ..., rule_n}
    * rule_i = {h_i, ¬b_i1, ..., ¬b_ik_i}
    * query = {¬q_1, ..., ¬q_m}     */
   if (m = 0) return true;
   for i := 1 to n {
      if((μ := Unify(q_1, h_i)) ≠ FAIL
         and SLDSolve(KB, {¬b_i1 μ, ..., ¬b_ik_i μ, ¬q_2 μ, ..., ¬q_m μ})) {
        return true;
        }
      }
      return false;
}
```

Where h_i, b_{ij}, and q_i are atomic formulae.

See B&L, p. 92

Example Prolog Interaction

```
<cirrus:Slides:1:101> sicstus
...
| ?- consult(user).
% consulting user...
| driver(X) :- drives(X,_).
| passenger(Y) :- drives(_,Y).
| drives(betty,tom).
| ^D
% consulted user in module user, 0 msec 776 bytes
yes
| ?- driver(X), passenger(Y).
X = betty,
Y = tom ?
yes
| ?-
```

Prolog Program with Two Answers

```
% From Rich & Knight, 2nd Edition (1991) p. 192.

likesToEat(X,Y) :- cat(X), fish(Y).
cat(X) :- calico(X).
fish(X) :- tuna(X).

tuna(charlie).
tuna(herb).
calico(puss).
```

Running the Fish Program

```
SICStus 3.9.1 (sparc-solaris-5.7): Thu Jun 27 22:58:18 MET DST 200
Licensed to cse.buffalo.edu
| ?- ['fish.prolog'].
% consulting /web/faculty/shapiro/UB/CSE563/Slides/fish.prolog...
% consulted /web/faculty/shapiro/UB/CSE563/Slides/fish.prolog
    in module user, 0 msec 1320 bytes
yes

| ?- likesToEat(puss,X).
X = charlie ? ;
X = herb ? ;
no

| ?- ^D
Process prolog finished
```

Tracing the Fish Program

```
| ?- ['fish.prolog'].
% consulting /web/faculty/shapiro/UB/CSE563/Slides/fish.prolog...
% consulted /web/faculty/shapiro/UB/CSE563/Slides/fish.prolog
    in module user, 0 msec 1320 bytes
yes

| ?- trace.
% The debugger will first creep -- showing everything (trace)
yes
% trace
```

Tracing First Answer

```
| ?- likesToEat(puss,X).
        1      1 Call: likesToEat(puss,_442) ?
        2      2 Call: cat(puss) ?
        3      3 Call: calico(puss) ?
        3      3 Exit: calico(puss) ?
        2      2 Exit: cat(puss) ?
        4      2 Call: fish(_442) ?
        5      3 Call: tuna(_442) ?
?       5      3 Exit: tuna(charlie) ?
?       4      2 Exit: fish(charlie) ?
?       1      1 Exit: likesToEat(puss,charlie) ?
X = charlie ? ;
```

Tracing the Second Answer

```
X = charlie ? ;
        1      1 Redo: likesToEat(puss,charlie) ?
        4      2 Redo: fish(charlie) ?
        5      3 Redo: tuna(charlie) ?
        5      3 Exit: tuna(herb) ?
        4      2 Exit: fish(herb) ?
        1      1 Exit: likesToEat(puss,herb) ?
X = herb ? ;
no

| ?- notrace.
% The debugger is switched off
yes
```

Negation by Failure & The Closed World Assumption

```
| ?- [user].
% consulting user...
| manager(jones, itSection).
| manager(smith, accountingSection).
|
% consulted user in module user, 0 msec 416 bytes
yes
| ?- manager(smith, itSection).
no
| ?- manager(kelly, accountingSection).
no
```

Negation by failure: “no” means didn’t succeed.
CWA: If it’s not in the KB, it’s not true.

Cut: Preventing Backtracking

KB Without Cut

```
| ?- consult(user).
% consulting user...
| bird(oscar).
| bird(tweety).
| bird(X) :- feathered(X).
| feathered(maggie).
| large(oscar).
| ostrich(X) :- bird(X), large(X).
| ^D
% consulted user in module user, 0 msec 184 bytes
yes
```

No Backtracking Needed

```
| ?- trace.
% The debugger will first creep -- showing everything (trace)
yes
% trace
| ?- ostrich(oscar).
        1      1 Call: ostrich(oscar) ?
        2      2 Call: bird(oscar) ?
?       2      2 Exit: bird(oscar) ?
        3      2 Call: large(oscar) ?
        3      2 Exit: large(oscar) ?
?       1      1 Exit: ostrich(oscar) ?
yes
```

Unwanted Backtracking

```
| ?- ostrich(tweety).
        1      1 Call: ostrich(tweety) ?
        2      2 Call: bird(tweety) ?
?       2      2 Exit: bird(tweety) ?
        3      2 Call: large(tweety) ?
        3      2 Fail: large(tweety) ?
        2      2 Redo: bird(tweety) ?
        4      3 Call: feathered(tweety) ?
        4      3 Fail: feathered(tweety) ?
        2      2 Fail: bird(tweety) ?
        1      1 Fail: ostrich(tweety) ?
no
```

No need to try to solve `bird(tweety)` another way.

KB With Cut

```
| ?- reconsult(user).
% consulting user...
| bird(oscar).
| bird(tweety).
| bird(X) :- feathered(X).
| feathered(maggie).
| large(oscar).
| ostrich(X) :- bird(X), !, large(X).
| ^D
% consulted user in module user, 0 msec -72 bytes
yes
```

Note `reconsult`.

No Extra Backtracking

```
| ?- trace.
% The debugger will first creep -- showing everything (trace)
yes
% trace
| ?- ostrich(tweety).
        1      1 Call: ostrich(tweety) ?
        2      2 Call: bird(tweety) ?
?       2      2 Exit: bird(tweety) ?
        3      2 Call: large(tweety) ?
        3      2 Fail: large(tweety) ?
        1      1 Fail: ostrich(tweety) ?
no
```

`fail`: Forcing Failure

If something is a canary, it is not a penguin.

```
| ?- reconsult(user).
% consulting user...
| penguin(X) :- canary(X), !, fail.
| canary(tweety).
| ^D
% consulted user in module user, 0 msec 352 bytes
yes
% trace
| ?- penguin(tweety).
        1      1 Call: penguin(tweety) ?
        2      2 Call: canary(tweety) ?
        2      2 Exit: canary(tweety) ?
        1      1 Fail: penguin(tweety) ?
no
```

`not`: "Negated" Antecedents

A bird that is not a canary is a penguin.

```
| penguin(X) :- bird(X), !, \+canary(X).
| bird(opus).
| ?- penguin(opus).
        1      1 Call: penguin(opus) ?
        2      2 Call: bird(opus) ?
        2      2 Exit: bird(opus) ?
        3      2 Call: canary(opus) ?
        3      2 Fail: canary(opus) ?
        1      1 Exit: penguin(opus) ?
yes
```

\+ is SICStus Prolog's version of `not`.
It is negation by failure, not logical negation.