

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`

copyright ©1995, 2004–2008 by Stuart C. Shapiro

Contents

1. Introduction	4
2. Propositional Logic	18
3. Predicate Logic Over Finite Models	135
4. Full First-Order Predicate Logic	181
5. Summary of Part I	297
6. Prolog	305
7. A Potpourri of Subdomains	335
8. SNePS	352
9. Belief Revision/Truth Maintenance	410
10. The Situation Calculus	462
11. Summary	481

Part III

12. Production Systems.....	494
13. Description Logic.....	503
14. Abduction.....	506

6 Prolog

6.1 Horn Clauses.....	306
6.2 Prolog	309

6.1 Horn Clauses

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

Either $\{\neg Q_1(\bar{x}), \dots, \neg Q_n(\bar{x})\}$ (negative Horn clause)

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

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

which is the same as

$$A_1(\bar{x}) \wedge \dots \wedge A_n(\bar{x}) \Rightarrow C(\bar{x})$$

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

SLD Resolution

Selected literals, Linear pattern, over Definite clauses

SLD derivation of clause c from set of clauses S is

$$c_1, \dots, c_n = c$$

$$\text{s.t. } c_1 \in S$$

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 = {rule1, ..., rulen}  
  * rulei = {hi, ¬bi1, ..., ¬biki}  
  * query = {¬q1, ..., ¬qm} */  
  if (m = 0) return true;  
  for i := 1 to n {  
    if((μ := Unify(q1, hi)) ≠ FAIL  
      and SLDSolve(KB, {¬bi1μ, ..., ¬bikiμ, ¬q2μ, ..., ¬qmμ})) {  
      return true;  
    }  
  }  
  return false;  
}
```

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

See B&L, p. 92

6.2 Prolog

Example Prolog Interaction

```
<pollux:CSE563:2:26> sicstus
...
| ?- consult(user).
% consulting user...
| driver(X) :- drives(X,_).
| passenger(Y) :- drives(_,Y).
| drives(betty,tom).
| ^D
% consulted user in module user, 0 msec 744 bytes
yes
| ?- driver(X), passenger(Y).
X = betty,
Y = tom ?
yes
| ?- halt.
```

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).

Listing the Fish Program

```
| ?- listing.  
calico(puss).
```

```
cat(A) :-  
    calico(A).
```

```
fish(A) :-  
    tuna(A).
```

```
likesToEat(A, B) :-  
    cat(A),  
    fish(B).
```

```
tuna(charlie).  
tuna(herb).
```

yes

Note: `consult(File)` loads the `File` in interpreted mode, whereas `[File]` loads the `File` in compiled mode. `listing` is only possible in interpreted mode.

Running the Fish Program

```
<pollux:CSE563:2:27> sicstus
SICStus 3.12.1 (sparc-solaris-5.7): Mon Apr 18 20:00:18 MET DST 2005
Licensed to cse.buffalo.edu
| ?- ['fish.prolog'].
% consulting /projects/shapiro/CSE563/fish.prolog...
% consulted /projects/shapiro/CSE563/fish.prolog in module user, 0 msec 135
yes

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

| ?- halt.
<pollux:CSE563:2:28>
```

Tracing the Fish Program

```
| ?- ['fish.prolog'].  
% consulting /projects/shapiro/CSE563/fish.prolog...  
% consulted /projects/shapiro/CSE563/fish.prolog in module user, 0 :  
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
```

```
% trace
```

```
| ?- 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).  
|  
% consulted user in module user, 0 msec 1120 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  
% trace
```

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

```
| ?- consult(user).  
% consulting user...  
| bird(oscar).  
| bird(tweety).  
| bird(X) :- feathered(X).  
| feathered(maggie).  
| large(oscar).  
| ostrich(X) :- bird(X), !, large(X).  
|  
% consulted user in module user, 0 msec -40 bytes  
yes  
% trace
```

No Extra 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) ?  
      1      1 Fail: ostrich(tweety) ?  
  
no  
% trace
```

fail: Forcing Failure

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

```
| ?- consult(user).
% consulting user...
| penguin(X) :- canary(X), !, fail.
| canary(tweety).
|
% consulted user in module user, 0 msec 416 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
% trace
```

Cut Fails the Head Instance: Program

```
penguin(X) :- canary(X), !, fail.  
penguin(X) :- bird(X), swims(X).
```

```
canary(tweety).  
bird(willy).  
swims(willy).
```

Cut Fails the Head Instance: Run

```
| ?- penguin(willy).  
    1      1 Call: penguin(willy) ?  
    2      2 Call: canary(willy) ?  
    2      2 Fail: canary(willy) ?  
    3      2 Call: bird(willy) ?  
    3      2 Exit: bird(willy) ?  
    4      2 Call: swims(willy) ?  
    4      2 Exit: swims(willy) ?  
    1      1 Exit: penguin(willy) ?
```

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

Cut Fails Head Alternatives

```
| ?- penguin(X).  
    1      1 Call: penguin(_368) ?  
    2      2 Call: canary(_368) ?  
    2      2 Exit: canary(tweety) ?  
    1      1 Fail: penguin(_368) ?
```

no

Moral:

Use cut when seeing if a ground atom is satisfied (T/F question),
but not when generating satisfying instances (wh questions).

Bad Rule Order

```
penguin(X) :- bird(X), swims(X).
penguin(X) :- canary(X), !, fail.
bird(X) :- canary(X).
canary(tweety).

% trace
| ?- penguin(tweety).
    1      1 Call: penguin(tweety) ?
    2      2 Call: bird(tweety) ?
    3      3 Call: canary(tweety) ?
    3      3 Exit: canary(tweety) ?
    2      2 Exit: bird(tweety) ?
    4      2 Call: swims(tweety) ?
    4      2 Fail: swims(tweety) ?
    5      2 Call: canary(tweety) ?
    5      2 Exit: canary(tweety) ?
    1      1 Fail: penguin(tweety) ?

no
```

Good Rule Order

```
penguin(X) :- canary(X), !, fail.  
penguin(X) :- bird(X), swims(X).  
bird(X) :- canary(X).  
canary(tweety).  
  
% trace  
| ?- penguin(tweety).  
      1      1 Call: penguin(tweety) ?  
      2      2 Call: canary(tweety) ?  
      2      2 Exit: canary(tweety) ?  
      1      1 Fail: penguin(tweety) ?  
  
no
```

SICSTUS Allows “or” In Body.

```
bird(willy).
swims(willy).
canary(tweety).
penguin(X) :-
    canary(X), !, fail;
    bird(X), swims(X).
bird(X) :- canary(X).

| ?- ['twoRuleCutOr.prolog'].
% consulting /projects/shapiro/CSE563/twoRuleCutOr.prolog...
* clauses for user:bird/1 are not together
* Approximate lines: 8-10, file: '/projects/shapiro/CSE563/twoRuleCutOr.prolog'
% consulted /projects/shapiro/CSE563/twoRuleCutOr.prolog in module user, 0 msec 1072 bytes
yes
| ?- penguin(willy).
yes
| ?- penguin(tweety).
no
```

not: “Negated” Antecedents

A bird that is not a canary is a penguin.

```
| penguin(X) :- bird(X), !, \+canary(X).  
| bird(opus).  
% consulted user in module user, 10 msec -40 bytes
```

```
| ?- 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.

Can Use Functions

```
driver(X) :- drives(X,_).  
drives(mother(X),X) :- schoolchild(X).  
schoolchild(betty).  
schoolchild(tom).
```

```
| ?- driver(X).  
X = mother(betty) ? ;  
X = mother(tom) ? ;  
no
```

Infinitely Growing Terms

```
driver(X) :- drives(X,_).
drives(mother(X),X) :- commuter(X).
commuter(betty).
commuter(tom).
commuter(mother(X)) :- commuter(X).

| ?- driver(X).
X = mother(betty) ? ;
X = mother(tom) ? ;
X = mother(mother(betty)) ? ;
X = mother(mother(tom)) ? ;
X = mother(mother(mother(betty))) ? ;
X = mother(mother(mother(tom))) ?
yes
```

Prolog Does Not Do the Occurs Check

```
<pollux:CSE563:2:31> sicstus
```

```
...
```

```
| ?- [user].
```

```
% consulting user...
```

```
| mother(motherOf(X), X).
```

```
|
```

```
% consulted user in module user, 0 msec 248 bytes
```

```
yes
```

```
| ?- mother(Y, Y).
```

```
Y = motherOf(motherOf(motherOf(motherOf(motherOf(motherOf(
    motherOf(motherOf(motherOf(motherOf(...)))))))))) ?
```

```
yes
```

```
| ?-
```

“=” and “is”

```
| ?- p(X, b, f(c,Y)) = p(a, U, f(V,U)).  
U = b,  
V = c,  
X = a,  
Y = b ?  
yes  
| ?- X is 2*(3+6).  
X = 18 ?  
yes  
| ?- X = 2*(3+6).  
X = 2*(3+6) ?  
yes  
| ?- X is 2*(3+6), Y is X/3.  
X = 18,  
Y = 6.0 ?  
yes  
| ?- Y is X/3, X is 2*(3+6).  
! Instantiation error in argument 2 of is/2  
! goal:  _76 is _73/3
```

Avoid Left Recursive Rules

To define `ancestor` as the transitive closure of `parent`.

The base case: `ancestor(X,Y) :- parent(X,Y).`

Three possible recursive cases:

1. `ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).`
2. `ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).`
3. `ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).`

Versions (2) and (3) will cause infinite loops.