KRR: Unit 3 Formative Activities

by Maria Ingold

Activity 1

Attempt the following questions from the module core text:

Q: Chapter 2 Question 4 - The barber's paradox.

4. In a certain town, there are the following regulations concerning the town barber:

Anyone who does not shave himself must be shaved by the barber.

Whomever the barber shaves, must not shave himself.

Show that no barber can fulfil these requirements. That is, formulate the requirements as sentences of FOL and show that in any interpretation where the first regulation is true, the second one must be false. (This is called the barber's paradox and was formulated by Bertrand Russell.)

A: Answer

First attempt:

- p = person
- b = barber
- A barber is a person
- Shaves(x,y)
- Shaves(b,p)
- Shaves(p,p) xor Shaves(b,p)
- Shaves(b,b) is impossible because
- Shaves(b,b) & not Shaves(p,p)

Realised here that predicate maps to true or false and a function maps to values. A predicate is not capitalised and a Function is capitalised.

- There exists person(x) = this person x exists (True or false)
- For all person(y) = For all the people
- shaves(x,y) = x shaves y (True or False)
    - y is either the same person as x ➔ shaves(x)
    - or y is the barber, but not both.
- but also for one x, shaves(x,y) implies not shaves(x,y)

(There exists an x such that) (person(x) and (for all y in person(y) implies shaves(x,y) but also that they don't shaves(x,y))

Can't be true.

Q: Chapter 3 Question 4 - A Canadian variant of an old puzzle

4. A Canadian variant of an old puzzle:

A traveler in remote Quebec comes to a fork in the road and does not know which way to go to get to Chicoutimi. Henri and Pierre are two local inhabitants nearby who do know the way. One of them always tells the truth, and the other one never does, but the traveler does not know which is which. Is there a single question the traveler can ask Henri (in French, of course) that will be sure to tell him which way to go?

We will formalize this problem in FOL. Assume there are only two sorts of objects in our domain: *inhabitants*, denoted by the constants henri and pierre; and *French questions*, which Henri and Pierre can answer. These questions are denoted by the following terms:

- gauche, which asks if the traveler should take the left branch of the fork to get to Chicoutimi;
- dit_oui(x, q), which asks if inhabitant x would answer yes to the French question q;
- dit_non(x, q), which asks if inhabitant x would answer no to the French question q.

Obviously this is a somewhat impoverished dialect of French, although a philosophically interesting one. For example, the term

dit_non(henri, dit_oui(pierre, gauche))

represents a French question that might be translated as, "Would Henri answer no if I asked him if Pierre would say yes I should go to the left to get to Chicoutimi?" The predicate symbols of our language are the following:

- Truth_teller(x), which holds when inhabitant x is a truth teller;
- Answer_yes(x, q), which holds when inhabitant x will answer yes to French question q;
- True(q), which holds when the correct answer to the question q is yes;
- Go_left, which holds if the direction to get to Chicoutimi is to go left.

For purposes of this puzzle, these are the only constant, function, and predicate symbols.

(a) Write FOL sentences for each of the following:

- *One of Henri or Pierre is a truth teller, and one is not.*
- *An inhabitant will answer yes to a question if and only if he is a truth teller and the correct answer is yes, or he is not a truth teller and the correct answer is not yes.*
- *The gauche question is correctly answered yes if and only if the proper direction is to go is left.*
- *A* dit_oui*(x, q) question is correctly answered yes if and only if x will answer yes to question q.*
- *A* dit_non*(x, q) question is correctly answered yes if and only if x will not answer yes to q.*

Imagine that these facts make up the entire KB of the traveler.

(b) Show that there is a ground term t such that

$$\text{KB} \models [\text{Answer\_yes}(\text{henri}, t) \equiv \text{Go\_left}].$$

In other words, there is a question t that can be asked to Henri (and there is an analogous one for Pierre) that will be answered yes if and only if the proper direction to get to Chicoutimi is to go left.

(c) Show that this KB does not entail which direction to go, that is, show that there is an interpretation satisfying the KB where Go_left is true, and another one where it is false.

A: Canadian Problem

(a) Write FOL sentences for each of the following:

One of Henri or Pierre is a truth teller, and one is not.

There exists only 1 x Truth_Teller(x) out of (henry, pierre)

There exists x, y such that: Truth_Teller(x) ^ ¬Truth_Teller(y)

An inhabitant will answer yes to a question if and only if he is a truth teller and the correct answer is yes, or he is not a truth teller and the correct answer is not yes.

For all x, q: Answer_yes(x,q) iff (TruthTeller(x) ^ True(q)) V (¬TruthTeller(x) ^ ¬True(q))

The gauche question is correctly answered yes if and only if the proper direction is to go is left.

For all x, q: Answer_yes(x, q) iff True(GoLeft)

Satisfy 4 conditions to express logic.

A dit_oui(x, q) question is correctly answered yes if and only if x will answer yes to question q.

For all x, q: True(dit_non(x,q)) iff Answer_yes(x,q)

A dit_non(x, q) question is correctly answered yes if and only if x will not answer yes to q.

For all x, q: True(dit_non(x,q)) iff ¬Answer_yes(x,q)

Imagine that these facts make up the entire KB of the traveller.

(b) Show that there is a ground term t such that

$$\mathrm{KB} \models [\mathrm{Answer\_yes}(\mathrm{henri}, t) \equiv \mathrm{Go\_left}].$$

In other words, there is a question t that can be asked to Henri (and there is an analogous one for Pierre) that will be answered yes if and only if the proper direction to get to Chicoutimi is to go left.

Answer_yes(truth_teller, will not truth teller tell me to go left)

If the answer is yes, then don't go left, which means right.

Answer_yes(not truth_teller, will truth teller tell me to go left)

If the answer is yes, then truth teller will not tell me to go left, which means right.

Answer_yes(truth_teller, will not truth teller tell me to go right?)

If the answer is yes, then answer is left.

Answer_yes(not truth_teller, will truth teller tell me to go right?)

If the answer is yes, then truth teller will not tell me to go right, which means left.

So, ask Will x tell me to go right?

Answer_yes(x, Will other tell me to go right?) ^ True(goLeft)

(c) Show that this KB does not entail which direction to go, that is, show that there is an interpretation satisfying the KB where Go_left is true, and another one where it is false.

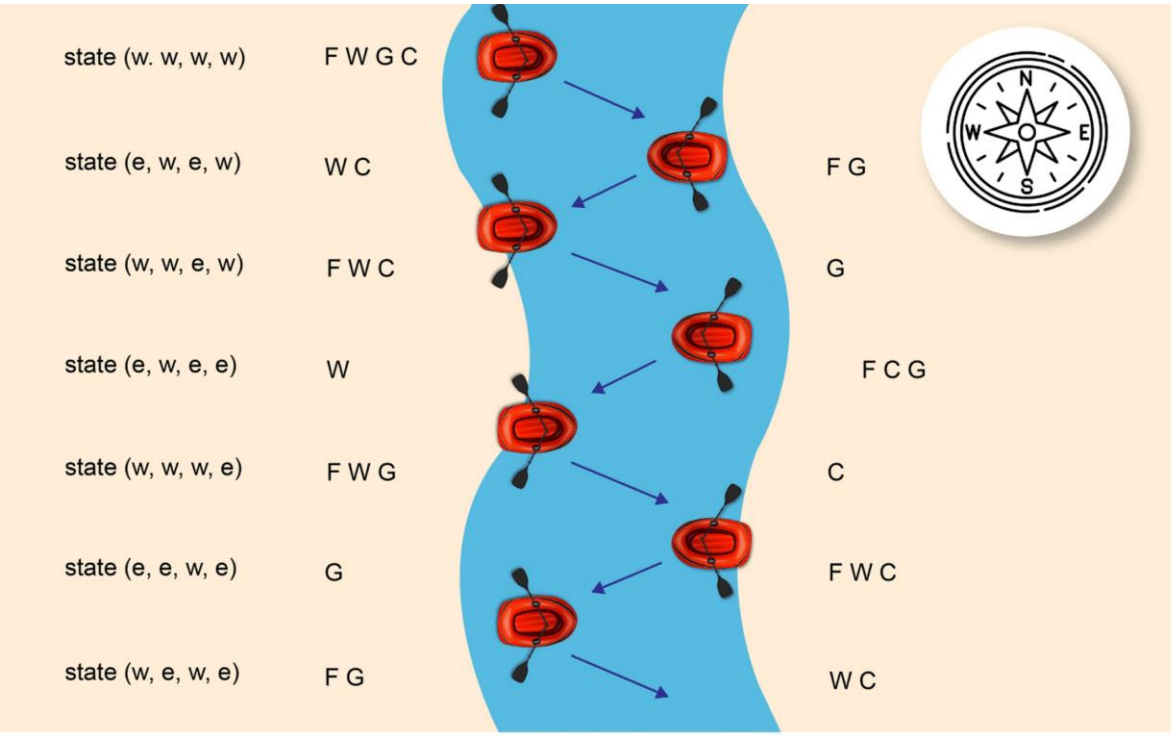
See above. If don't know if True(goLeft)

- True(goLeft) ^ Answer_yes(x, Will other tell me to go right?)
- True(goLeft) ^ Answer_yes(x, Will other tell me to go left?)

Activity 2

Q: The Crossing Problem

Read the paper by Palomino et al (2005) and review the 'crossing problem' diagram provided in the Lecturecast.



A: The Crossing Problem

Q: Create a set of statements in first order logic (FOL) that represent the states shown – for example you may define two functions left and right and therefore the first state could be represented as: Left(F) and left(W) and Left(G) and left(C). Define your own set of FOL statements for the entire diagram.

- Left = w side of river (Left(x))
- Right = e side of river (Right(x))
- initial = w (Left(x))
- shepherd = S
- goat = G
- cabbage = C
- wolf = W
- Rules:
  - If ¬shepherd ^ wolf ^ goat → Fail
  - If ¬shepherd ^ goat ^ cabbage → Fail
  - Maude said If ¬shepherd ^ ¬wolf ^ goat ^ cabbage → Fail
- Implied Rules
  - If ¬shepherd ^ wolf ^ cabbage → Succeed
  - If ¬shepherd ^ wolf → Succeed
  - If ¬shepherd ^ goat → Succeed
  - If ¬shepherd ^ cabbage → Succeed
  - If shepherd → Succeed

State	shepherd	wolf	goat	cabbage	Left/west	Right/east
State (location)	w	w	w	w	SWGC	NO ONE
Take goat	e	w	e	w	WC	SG
Leave goat	w	w	e	w	SWC	G
Take cabbage	e	w	e	e	W	SGC
Leave cabbage/return goat	w	w	w	e	SWG	C
Leave goat west / take wolf to cabbage	e	e	w	e	G	SWC
Go back to goat / leave wolf/cabbage	w	e	w	e	SG	WC
Take goat and get to east side with everyone intact	e	e	e	e	NO ONE	SWGC

- Left(S) ^ Left(W) ^ Left(G) ^ Left(C)
- Right(S) ^ Left(W) ^ Right(G) ^ Left(C)
- Left(S) ^ Left(W) ^ Right(G) ^ Left(C)
- Right(S) ^ Left(W) ^ Right(G) ^ Right (C)
- Left(S) ^ Left(W) ^ Left(G) ^ Right (C)
- Right(S) ^ Right(W) ^ Left(G) ^ Right (C)
- Left(S) ^ Right(W) ^ Left(G) ^ Right (C)
- Right(S) ^ Right(W) ^ Right(G) ^ Right (C)

Q: You have been provided with solutions to the ‘crossing problem’ written in Lisp (here), Prolog (here) and Maude (here). Compare your FOL clauses with the various implementations – which of the implementations provides the closest match to your FOL version?

Links were missing. Think it is Prolog. These were the lecturecast examples:

Example 1: LISP

```
;;; This is the farmer, wolf, goat and cabbage problem from section 7.2
;;; of the text.
;;; solve-fwgc initiates the search.  A typical starting function call
;;; might be:
;;;
;;;                    (solve-fwgc '(e e e e) '(w w w w))
;;;
;;; This finds a path from the east bank to the west.
(defun solve-fwgc (state goal) (path state goal nil))

;;; The recursive path algorithm searches the space in a depth first
;;; fashion.

(defun path (state goal been-list)
   (cond ((null state) nil)
   ((equal state goal) (reverse (cons state been-list)))
   ((not (member state been-list :test #'equal))
        (or (path (farmer-takes-self state) goal (cons state been-list))
            (path (farmer-takes-wolf state) goal (cons state been-list))
            (path (farmer-takes-goat state) goal (cons state been-list))
            (path (farmer-takes-cabbage state) goal (cons state been-list))))))

;;; These functions define legal moves in the state space.  The take
;;; a state as argument, and return the state produced by that operation.

(defun farmer-takes-self (state)
   (safe (make-state (opposite (farmer-side state))
    (wolf-side state)
    (goat-side state)
    (cabbage-side state))))

(defun farmer-takes-wolf (state)
   (cond ((equal (farmer-side state) (wolf-side state))
                     (safe (make-state (opposite (farmer-side state))
                                       (opposite (wolf-side state))
                                       (goat-side state)
                                       (cabbage-side state))))
         (t nil)))
```

Example 2: Prolog

```
path(State,Goal,Been_stack) :-
  move(State,Next_state),
  not(member_stack(Next_state,Been_stack)),
  stack(Next_state,Been_stack,New_been_stack),
  path(Next_state,Goal,New_been_stack),!.

/*
 * Move predicates
 */

move(state(X,X,G,C), state(Y,Y,G,C))
              :- opp(X,Y), not(unsafe(state(Y,Y,G,C))),
                 writelist(['try farmer takes wolf',Y,Y,G,C]).

move(state(X,W,X,C), state(Y,W,Y,C))
              :- opp(X,Y), not(unsafe(state(Y,W,Y,C))),
                 writelist(['try farmer takes goat',Y,W,Y,C]).

move(state(X,W,G,X), state(Y,W,G,Y))
              :- opp(X,Y), not(unsafe(state(Y,W,G,Y))),
                 writelist(['try farmer takes cabbage',Y,W,G,Y]).

move(state(X,W,G,C), state(Y,W,G,C))
              :- opp(X,Y), not(unsafe(state(Y,W,G,C))),
                 writelist(['try farmer takes self',Y,W,G,C]).

move(state(F,W,G,C), state(F,W,G,C))
              :- writelist(['      BACKTRACK from:',F,W,G,C]), fail.

/*
 * Unsafe predicates
 */

unsafe(state(X,Y,Y,C)) :- opp(X,Y).
unsafe(state(X,W,Y,Y)) :- opp(X,Y).

/*
 * Definitions of writelist, and opp.
```

Example 3: Maude

```
4   mod RIVER-CROSSING is
5     sorts Side Group State .
6     ops left right : -> Side [ctor] .
7     op change : Side -> Side .
8     --- shepherd, wolf, goat, cabbage
9     ops s w g c : Side -> Group [ctor] .
10    op __ : Group Group -> Group [ctor assoc comm] .
11    op {_} : Group -> State [ctor] .
12    op toBeEaten : Group -> Bool .
13    op initial : -> State .
14    vars S S' : Side .
15    var  G : Group .
16    eq change(left) = right .
17    eq change(right) = left .
18    ceq w(S) g(S) s(S') = w(S) s(S') if S =/= S' .
19        --- wolf eats goat
20    ceq c(S) g(S) w(S') s(S') = g(S) w(S') s(S') if S =/= S' .
21        --- goat eats cabbage
22    ceq toBeEaten(w(S) g(S) s(S') G) = true if S =/= S' .
23    ceq toBeEaten(c(S) g(S) s(S') G) = true if S =/= S' .
24    eq toBeEaten(G) = false [owise] .
25
26    eq initial = { s(left) w(left) g(left) c(left) } .
27
28    crl [shepherd-alone] : { s(S) G } => { s(change(S)) G }
29      if not(toBeEaten(s(S) G)) .
30    crl [wolf] : { s(S) w(S) G } => { s(change(S)) w(change(S)) G }
31      if not(toBeEaten(s(S) w(S) G)) .
32    rl [goat] : { s(S) g(S) G } => { s(change(S)) g(change(S)) G }.
33    crl [cabbage] : { s(S) c(S) G } => { s(change(S)) c(change(S)) G }
34      if not(toBeEaten(s(S) c(S) G)) .
35  endm
```