Situation Calculus: Door Example

(reminder) Steps to Axiomatize a Domain

In the following steps, the word "determine" implies "write down"

1. Understand the domain by reading about it, studying it, and thinking about it
2. Determine the set of fluents that are sufficient to represent a state in the domain
3. Determine the set of actions that effect (bring about) change in the state (fluent truth values)
4. Determine the precondition axioms for actions in terms of fluents
5. Determine the successor state axiom for each fluent
6. Determine the fluent values in the initial situation s0 (we use [] for s0).

Door example

Let's consider a simple example of closing or opening a door. (Step 1)

What represents the state of the world? What can be true or false? (Step 2)

Fluent list: the special (fluent) predicates we need to make statements (Step 2)

near_door(s): true if the robot is near the door "in s"

door_open(s): true if the door is open "in s"

Do we need a door_closed(s) fluent? Or can we use “not door_open(s)”

What can our robot do? (Step 3)

Action list: go_to_door, open_door, close_door (Step 3)

Axiomatize door world

After determining what the actions are, and what fluents are needed to characterize a state (previous slide), then

- We need a set of precondition axioms, one for each action, to specify what can happen in our simple world (Step 4)
- We need a set of successor state axioms, one for each fluent, to specify what changes and what stays the same when a possible action happens (Step 5)
- We need a set of initial state axioms, one for each fluent, to specify the truth value of the fluent in the initial situation, [], before anything happens (Step 6)

Precondition axioms for door

We will write our axioms directly in clausal form, using Prolog notation, where variables are capitalized. Negation \+ causes problems that we'll revisit later.

Notice this code uses the two-argument syntax for poss.

```
poss(go_to_door,S):- \+ near_door(S).

poss(leave_door,S):- near_door(S).

poss(open_door,S):- near_door(S), \+ door_open(S).

poss(close_door,S):- near_door(S), door_open(S).
```

What would be the equivalent code for the one-argument syntax for poss?

```
poss([go_to_door|S]):-\+ near_door(S).   etc.
```

Successor State axioms for door

Again, we use Prolog notation.

% door_open is true after we open the door; or, if it was already open and we didn't close it

```
door_open(do(A,S)):-      % two arguments. One argument: door_open([A|S]):-
    poss(A,S),        % two arguments.  One argument: poss([A|S])
    ( A = open_door
      ;
     A \= close_door,
     door_open(S)
    ).
```

Successor State axioms for door (cont'd)

```
% we are near the door after we go to the door; or, if we were
already near the door (and we didn't go away – leave_door).

near_door(do(A,S):-

    poss(A,S), % two arguments.  One argument poss([A|S])

    (A = go_to_door

     ;

    near_door(S),

    A \= leave_door

    ).
```

Successor State axioms for door, again

In prolog, A:-B;C can be written as two clauses

A:-B.

A:-C.

Right? Think about it… So, we have

```
door_open(do(open_door,S)):- poss(open_door,S).
door_open(do(A,S)):- poss(A,S), A \= close_door, door_open(S).
near_door(do(go_to_door,S)):- poss(go_to_door,S).
near_door(do(A,S):- poss(A,S), A \= leave_door, near_door(S).
```

Thinking about the previous slide

A:-B;C

(B or C)->A

not (B or C) or A

(not B and not C) or A

(not B or A) and (not C or A)

B->A and C->A

Now, in Prolog

A:-B.

A:-C.

Initial State for door domain

For our very simple door domain

- We have a set of actions, and precondition axioms for those
- We have a set of fluents, and successor state axioms for those
- We also need to specify the initial state:
    - near_door(S_0) : true or false?
    - door_open(S_0) : true or false?

Situation observations revisited

If **a** and **b** are actions, and if we consider the empty list **[ ]** to be like $\mathbf{S_0}$, then

do(b,do(a,$\mathbf{S_0}$)) can be represented in prolog as **[b,a]**

We will do this.

Recall that a list is either the atom **[ ]**, or of the form **[H|T]**, and in dot functor notation, **[H|T]** is represented as **.(H,T)**

So, **[b,a]** is equivalent to **.(b,.(a,[ ]))**

See the parallel between "**do**" and "**.**", and "$\mathbf{S_0}$" and "**[ ]**"

Prolog initial state

Suppose our initial state is that the door is closed, and we are not near the door:

near_door([ ]) is false

door_open([ ]) is false

We have a negation problem. In prolog we specify these are false by omitting them, so our initial state is specified by saying nothing about the initial state! This is a problem!

Saying nothing results in this error instead of the expected "false":

```
?- door_open([]).

ERROR: Undefined procedure: door_open/1 (DWIM could not correct goal)

?-
```

Avoiding negation

We want door_open([ ]), to be false, but the following prolog representation is lacking

```
door_open([ ]):-false.
```

With this representation, we get

```
?- door_open([ ]).
false
```

which is half of what we want. Still have a problem…

Avoiding negation (cont'd)

The other half of what we want is to find situations where the door is not open, and we know that is true of [ ]:

With this representation, we get

```
?- \+ door_open(S).     % is there an S such that door_open(S) is false?
false
```

which fails also, even though we know it is true for []?

Negation problem: Goals that fail do not result in variable instantiation!

Avoiding negation (solving the negation problem)

We can represent this type of boolean-valued fluent explicitly by adding an argument to represent the truth value.

door_open(y,S) means the door is open in S

door_open(n,S) means the door is not open in S:

```
door_open(n,[ ]).     % \+ door_open([ ]).

?- door_open(n,S).    % find situations in which \+ door_open(S)

S=[ ]
```

Running the door program

Let's write down the axioms for the door program, in prolog:

To start, here is the initial state:

```
door_open(n,[ ]).

near_door(n,[ ]).
```

Complete Prolog for door example

%For a lab or assignment, you would be expected to 1) include a header comment stating the purpose of the program 2) include a declarative comment for each predicate stating what the predicate means in terms of its arguments, 3) properly format each rule with proper indentation

```
door_open(n,[]).

door_open(y,[open_door|S]):- poss([open_door|S]).

door_open(y,[A|S]):- poss([A|S]), A \= close_door, door_open(y,S).

door_open(n,[close_door|S]):- poss([close_door|S]).

door_open(n,[A|S]):- poss([A|S]), A \= open_door, door_open(n,S).

near_door(n,[]).

near_door(y,[go_to_door|S]):- poss([go_to_door|S]).

near_door(y,[A|S]):- poss([A|S]),  A \= leave_door, near_door(y,S).

near_door(n,[leave_door|S]):- poss(l[leave_door|S).

near_door(n,[A|S]):- poss([A|S]),  A\= go_to_door, near_door(n,S).

poss([go_to_door|S]):- near_door(n,S).

poss([leave_door|S]):- near_door(y,S).

poss([open_door|S]):- near_door(y,S), door_open(n,S).

poss([close_door|S]):- near_door(y,S), door_open(y,S).
```

Conclusion

In this lesson, you learned how to represent knowledge of domains in the situation calculus, and how to translate that into practical Prolog programs.

In the next lesson, you will learn to solve planning problems using the situation calculus and Prolog.