

1 Situation Calculus Terminology Review

When we first started discussing first-order logic (FOL) knowledge bases, we were using them to representing properties of the world without a notion of time. For example, whether a pelican is a type of bird does not change with time. We will now describe the use of FOL knowledge bases for representing properties of the world that change in time. For example, we would like to represent that Mrs. Rabbit did not have a loaf of brown bread before she bought it, but Mrs. Rabbit did have a loaf of brown bread after she bought it.

To represent properties of the world that change in time, we represent the world as a sequence of states or situations, which are snapshots of the world at some instant. In each state, some properties of the world hold, and some do not. To represent how the properties of the world change from one state to the next, we state which properties are caused and canceled by performing a particular action in some initial state. For example, in the initial state, Mrs. Rabbit does not have a loaf of brown bread. Performing the action of buying a loaf of brown bread causes Mrs. Rabbit having a loaf of brown bread. So, in the state resulting from performing the action of buying a loaf of brown bread in the initial state, Mrs. Rabbit does have a loaf of brown bread.

The *situation calculus* is the common representation of actions and their results using FOL. In the situation calculus, there are three types of symbols, each represented by a term in FOL:

- A *state* is a snapshot of the world.
- A *fluent* is a property of the world that changes with time. For example, `HasLoafOfBrownBread(MrsRabbit)` is a fluent.
- An *action* is an event that changes one state into another. For example, `BuyLoafOfBrownBread(MrsRabbit)` is an action.

There are four special symbols that are used in the situation calculus:

- `holds` is a predicate which connects fluents with states. It has two input terms: a fluent and a state. For example, the sentence `holds(HasLoafOfBrownBread(MrsRabbit), s1)` could be used to express that Mrs. Rabbit has a loaf of brown bread at state `s1`.
- `do` is a function which returns the state resulting from doing an action in an initial state. It takes two inputs: an state and an action, and returns one output state. For example, the sentence:
`s1 = do(s0, BuyLoafOfBrownBread(MrsRabbit))`
could be used to express that `s1` is the state resulting from buying a loaf of brown bread in state `s0`.
- `causes` is a predicate used to describe what fluents hold in the next state as a result of performing an action in an initial state. It takes three inputs: an action, an initial state, and a fluent. For example, the sentence
`causes(BuyLoafOfBrownBread(MrsRabbit), s0, HasLoafOfBrownBread(MrsRabbit))`
could be used to express that the action `BuyLoafOfBrownBread(MrsRabbit)` in the initial state `s0` causes the fluent `HasLoafOfBrownBread(MrsRabbit)` to hold.
- `cancels` is a predicate used to describe what fluents do *not* hold in the next state as a result of performing an action in and initial state. It takes three inputs: an action, an initial state, and a fluent.

Specifying that only fluents that are caused or canceled by an action change from one state to the next when an action is performed is called the *frame problem*. This knowledge is represented by the following axiom:

$$\forall \text{action, state, fluent} \quad \text{holds}(\text{fluent}, \text{do}(\text{state}, \text{action})) \leftrightarrow \text{causes}(\text{action}, \text{state}, \text{fluent}) \vee (\text{holds}(\text{fluent}, \text{state}) \wedge \neg \text{cancels}(\text{action}, \text{state}, \text{fluent})).$$

2 Introduction to Otter

Otter (available at <http://www-unix.mcs.anl.gov/AR/otter/>) is a system that proves sentences stated in FOL. It takes input from stdin and outputs to stdout. To run it from the UNIX command prompt, use the following command:

```
otter < [input file] > [output file]
```

Here is a skeleton input file:

```
% Comment
set(auto).

formula_list(usable).

% All well-formed formulas go here.

end_of_list.
```

A well-formed formula is a sentence in FOL. Each sentence must end with a period. Here are some of the special symbols available in Otter:

FOL	Otter
\wedge (and)	&
\vee (or)	
\neg (not)	-
\rightarrow (implication)	- >
\leftrightarrow (equivalence)	< - >
\exists (there exists)	exists
\forall (for all)	all

Otter also has the integers and integer operations built into it. Otter also allows you to use the symbol “=” (with opposite “!=”) in infix notation. However, *it does not have the definition of = built in*. Thus, to use = as we normally do, you must have the axiom: `all x (x = x).`

Note that Otter also does not have a unique names assumption, so it does not know that `3 = 4` is false.

Otter tells variables from constant symbols based on the first letter of the symbol. If the name begins with lowercase u, v, w, x, y, or z, it is assumed to be a variable. Otherwise, it is assumed to be a constant.

Otter tries to prove that there is a contradiction in the input sentences. For example, if we input

```
Rabbit(MrsRabbit).
-Rabbit(MrMacGregor).
MrMacGregor = MrsRabbit.
```

Otter will prove that there is a contradiction. Here is the output it produces:

```
----- PROOF -----
1 [] -Rabbit(MrMacGregor).
2 [] Rabbit(MrsRabbit).
3 [] MrMacGregor=MrsRabbit.
5,4 [copy,3,flip.1] MrsRabbit=MrMacGregor.
6 [back_demod,2,demod,5] Rabbit(MrMacGregor).
7 [binary,6.1,1.1] $F.
```

```
----- end of proof -----
```

Otter proves that, if `MrsRabbit = MrMacGregor`, then `Rabbit(MrMacGregor)`, which contradicts the sentence `-Rabbit(MrMacGregor)`. Suppose, then, that our knowledge base is just the sentences:

```
Rabbit(MrsRabbit).
-Rabbit(MrMacGregor).
```

and we want to use Otter to prove that it is necessarily true that `MrMacGregor != MrsRabbit`. We can do this by introducing the sentence `MrMacGregor = MrsRabbit`, and proving a contradiction.

3 Encoding a Narrative in Otter

When you encode the Peter Rabbit narrative, there are three steps you should take:

1. Encode relevant knowledge about the world. This includes defining your ontology as well as encoding commonsense knowledge about the relationships between the symbols. Defining these relationships requires describing how actions change the properties of the world, using the causes-cancels framework. Also, it requires describing the relationship between fluents. It is important to do a complete job of encoding knowledge about the world, so that we can infer properties of the world at different states.
2. Encode the specific events that happened in the narrative.
3. Encode some queries about either the world or the narrative.

For example, consider the following narrative:

Mary is at home, and standing outside her car, holding her car key. She unlocks the car door, opens it, and gets in the car. She closes the door, puts her key in the ignition, starts the car, and drives to the baker's.

We begin by representing the relevant knowledge about the world in a simple but general way. First, we determine what objects are in the world. We choose the following constants:

Constant Symbol	Interpretation
Mary	The person Mary
Home	Mary's home.
CarKey	Mary's car key
Car	Mary's car
Bakers	The baker's store.

Next, we determine the properties of the world that change in time. We choose the following fluents:

Fluent	Interpretation
IsAt(x, y)	Object x is at location y.
IsNear(x)	Mary is near object x.
IsHolding(x)	Mary is holding object. x.
IsLocked(x)	The door of object x is locked.
IsOpen(x)	The door of object x is open.
KeyIsInIgnition	Mary's key is in the ignition.
CarIsRunning	Mary's car is running.
IsIn(x)	Mary is in object x.

We define one property that does not change with time, if object x is the key for object y, which is represented by a FOL predicate $IsKeyTo(x, y)$.

We now represent how the world changes from one state to the next. We choose the following actions:

Action	Interpretation
Unlocks(x, y)	Mary unlocks the door of object x with object y.
Locks(x, y)	Mary locks the door of object x with object y.
Opens(x)	Mary opens the door of object x
Closes(x)	Mary closes the door of object x
GetsIn(x)	Mary gets into object x.
GetsOut(x)	Mary gets out of object x.
PutsKeyInIgnition	Mary puts her car key in the ignition.
StartsCar	Mary starts her car.
Drives(x, y)	Mary drives her car from location x to location y.

We must define the effects of performing these actions in some initial state. We do this using the causes and cancels predicates. We formalize our commonsense about the actions above:

- In order for Mary to unlock the door of object x with object y, she must be near object x, must be holding object y, and object y must be the key to object x. If these hold, then after the action is performed, the door is not locked. We formalize this as:
 $all\ u\ x\ y\ ((IsKeyTo(y, x) \ \&\ holds(IsNear(x), u) \ \&\ holds(IsHolding(y), u)) \rightarrow cancels(Unlocks(x, y), u, IsLocked(x)))$.
- To lock the door of object x with object y, Mary must be near the object, holding object y, and object y must be the key to object x. If this is true, then after the action is performed the door is locked. We formalize this as:
 $all\ u\ x\ y\ ((IsKeyTo(y, x) \ \&\ holds(IsNear(x), u) \ \&\ holds(IsHolding(y), u)) \rightarrow causes(Locks(x), u, IsLocked(x)))$.
- In order for Mary to open the door of object x, it must be unlocked and Mary must be near object x. If this is true, then after the action is performed the door is open. We formalize this as:
 $all\ u\ x\ ((holds(IsNear(x), u) \ \&\ \neg holds(IsLocked(x), u)) \rightarrow causes(Opens(x), u, IsOpen(x)))$.
- In order for Mary to close the door of object x, Mary must be near object x. If this is true, then after the action is performed the door is closed. We formalize this as:
 $all\ u\ x\ (holds(IsNear(x), u) \rightarrow cancels(Closes(x), u, IsOpen(x)))$.

- In order for Mary to get in object x , Mary must be near x and the door of x must be open. If this is the case, then after the action is performed, Mary is in the object. We formalize this as:
 $\text{all } u \ x \ ((\text{holds}(\text{IsNear}(x), u) \ \& \ \text{holds}(\text{IsOpen}(x), u)) \rightarrow \text{causes}(\text{GetsIn}(x), u, \text{IsIn}(x)))$.
- In order for Mary to get out of object x , Mary must be in x and the door of x must be open. If this is the case, then after the action is performed, Mary is in the object. We formalize this as:
 $\text{all } u \ x \ ((\text{holds}(\text{IsIn}(\text{Mary}, x), u) \ \& \ \text{holds}(\text{IsOpen}(x), u)) \rightarrow \text{cancels}(\text{GetsOut}(x), u, \text{IsIn}(x)))$.
- In order for Mary to put her key in the ignition, her key must be the car key, Mary must be holding the key, and Mary must be in the car. If this is the case, then after action is performed, the key is in the ignition and Mary is no longer holding the key. We formalize this as:
 $\text{all } u \ x \ ((\text{IsKeyTo}(x, \text{Car}) \ \& \ \text{holds}(\text{IsIn}(\text{Car}), u) \ \& \ \text{holds}(\text{IsHolding}(x), u)) \rightarrow (\text{causes}(\text{PutsKeyInIgnition}, u, \text{KeyIsInIgnition}) \ \& \ \text{cancels}(\text{PutsKeyInIgnition}, u, \text{IsHolding}(x))))$.
- In order for Mary to start her car, she must be in the car and her key must be in the ignition. If this is the case, then after the action is performed, the car is running. We formalize this as:
 $\text{all } u \ x \ ((\text{holds}(\text{IsIn}(\text{Car}), u) \ \& \ \text{holds}(\text{KeyIsInIgnition}, u)) \rightarrow \text{causes}(\text{StartsCar}, u, \text{CarIsRunning}))$.
- In order for Mary to drive her car from location x to location y , she must be at x and in her car, her car must be running, and the car door must be closed. If this is the case, then after the action is performed, Mary is at location y . We formalize this as:
 $\text{all } u \ x \ y \ ((\text{holds}(\text{IsAt}(\text{Mary}, x), u) \ \& \ \text{holds}(\text{IsIn}(\text{Car}), u) \ \& \ \text{holds}(\text{CarIsRunning}, u) \ \& \ \text{holds}(\neg \text{IsOpen}(\text{Car}), u)) \rightarrow (\text{causes}(\text{Drives}(x, y), u, \text{IsAt}(\text{Mary}, y)) \ \& \ \text{cancels}(\text{Drives}(x, y), u, \text{IsAt}(\text{Mary}, x))))$.

In order to avoid typing problems, we combine all of these causes-cancels statements into two axioms, and use an equivalence instead of an implication, as in the lecture notes.

We must also represent relationships between fluents. For example, if Mary is in object x then she is near it:

$\text{all } u \ x \ (\text{holds}(\text{IsIn}(x), u) \rightarrow \text{holds}(\text{IsNear}(x), u))$. Also, if Mary is at location y and she is near object x , then object x is also at location y :

$\text{all } u \ x \ y \ ((\text{holds}(\text{IsAt}(\text{Mary}, y) \ \& \ \text{holds}(\text{IsNear}(x), u)) \rightarrow \text{holds}(\text{IsAt}(x, y), u))$.

Once we have encoded sufficient knowledge about the world, we can encode the events that happen in the narrative:

- Mary's car is locked:
 $\text{holds}(\text{IsLocked}(\text{Car}), s_0)$.
- Mary's car is not running:
 $\neg \text{holds}(\text{CarIsRunning}, s_0)$.
- Mary's car door is closed.
 $\neg \text{holds}(\text{IsOpen}(\text{Car}), s_0)$.
- Mary is at home:
 $\text{holds}(\text{IsAt}(\text{Mary}, \text{Home}), s_0)$.
- standing outside her car:
 $\text{holds}(\text{IsNear}(\text{Car}), s_0)$.
- and holding her car key:
 $\text{holds}(\text{IsHolding}(\text{CarKey}), s_0)$.
 $\text{IsKeyTo}(\text{CarKey}, \text{Car})$.
- She unlocks the car door:
 $s_1 = \text{do}(s_0, \text{Unlocks}(\text{Car}, \text{CarKey}))$.
- opens it:
 $s_2 = \text{do}(s_1, \text{Opens}(\text{Car}))$.
- and gets in the car:
 $s_3 = \text{do}(s_2, \text{GetsIn}(\text{Car}))$.
- She closes the door:
 $s_4 = \text{do}(s_3, \text{Closes}(\text{Car}))$.
- puts her key in the ignition:
 $s_5 = \text{do}(s_4, \text{PutsKeyInIgnition})$.
- starts the car:
 $s_6 = \text{do}(s_5, \text{StartsCar})$.
- and drives to the baker's:
 $s_7 = \text{do}(s_6, \text{Drive}(\text{Home}, \text{Bakers}))$.

We can now make queries to our database. For instance, we could ask, where is Mary now? We would encode this question by:
`-(exists x holds(IsAt(Mary,x),s7)).`

Here, we are writing a sentence that produces a contradiction with the narrative sentences. Otter will find this contradiction by deducing that at state `s7`, Mary is at the Baker's:

`holds(IsAt(Mary,Bakers),s7).`

We could also ask a question like, did Mary open her car door? You might try to encode this sentence as:

`-(exists u1 u2 (u2 = do(u1, Opens(Car)))).`

Since there is the sentence:

`s2 = do(s1, Opens(Car))`

in our narrative, Otter will be able to prove a contradiction.

However, Otter could never prove a negative response to this question. That is, you could never enter the sentence:

`exists u1 u2 (u2 = do(u1, Locks(Car)))`

and have Otter prove that it never happened that Mary did not lock her door during the narrative. Instead, this question is asking if it is possible for Mary to lock her door. In order to answer questions about the narrative like did Mary lock her door, we will need to add to the situation calculus language. We could add a predicate `actual(s)` which is true only for the input states `s` that are mentioned in the narrative. Then, we could ask the question:

`exists u1 u2 (actual(u1) & actual(u2) & (u2 = do(u1, Locks(Car))))`

and prove a contradiction.

In order to encode longer narratives, it would be nice if we didn't have to explicitly encode all the intermediate steps. For instance, in the narrative above, we would understand what happened, if our narrative just consisted of:

Mary's car is locked and is not running. Mary gets in her car and drives to the baker's.

Since we know that getting in the car requires unlocking and opening the door, we don't explicitly state that Mary did these actions. In fact, our definitions of the actions are strong enough to make these inferences. The only problem we have is that we can only describe the relationship between consecutive states. That is, we can say that first we did `action1` and then we directly did `action2` (`s2 = do(s1, action1)` and `s3 = do(s2, action2)`). We cannot say that first we did `action1`, then we might have done some other actions, then we did `action2`. In order to encode this looser ordering, we introduce a new predicate into our situation calculus language, `after(s2,s1)`, with the meaning that state `s2` occurred some time after state `s1`. We need to define the meaning of the `after` predicate:

`all u2 u1 (after(u2,u1) <-> ((exists u_action (u2 = do(u1,u_action))) | (exists u3 ((u1 != u3) & (u2 != u3) & after(u2,u3) & after(u3,u1)))).`

Then, we could encode the simpler narrative:

- Mary's car is locked:
`holds(IsLocked(Car),s0).`
- Mary's car is not running:
`-holds(CarIsRunning,s0).`
- Mary's car door is closed.
`-holds(IsOpen(Car),s0).`
- Mary is at home:
`holds(IsAt(Mary,Home),s0).`
- Mary gets in her car: `s2 = do(s1, GetsIn(Car)).`
`after(s1,s0).`
- and drives to the baker's:
`s4 = do(s3, Drives(Home,Bakers)).`
`after(s3,s2).`