Artificial Intelligence I

Dr Sean Holden

Notes on *knowledge representation and reasoning using first-order logic (FOL)*

Copyright © Sean Holden 2002-2010.

Knowledge representation and reasoning using FOL

We now look at how an agent might *represent* knowledge about its environment using first order logic (FOL), and *reason* with this knowledge to achieve its goals.

Aims:

- To show how FOL can be used to *represent knowledge* about an environment in the form of both *background knowledge* and *knowledge derived from percepts*.
- To show how this knowledge can be used to *derive non-perceived knowledge* about the environment using a *theorem prover*.
- To introduce the *situation calculus* and demonstrate its application in a simple environment as a means by which an agent can work out what to do next.

Interesting reading

Reading: Russell and Norvig, chapters 7 to 10.

Knowledge representation based on logic is a vast subject and can't be covered in full in the lectures.

In particular:

- Techniques for representing *further kinds of knowledge*.
- Techniques for moving beyond the idea of a *situation*.
- Reasoning systems based on *categories*.
- Reasoning systems using *default information*.
- *Truth maintenance systems*.

Happy reading :-)

Knowledge representation and reasoning

Earlier in the course we looked at what an *agent* should be able to do.

It seems that all of us—and all intelligent agents—should use *logical reasoning* to help us interact successfully with the world.

Any intelligent agent should:

- Possess *knowledge* about the *environment* and about *how its actions affect the environment*.
- Use some form of *logical reasoning* to *maintain* its knowledge as *percepts* arrive.
- Use some form of *logical reasoning* to *deduce actions* to perform in order to achieve *goals*.

Knowledge representation and reasoning

This raises some important questions:

- How do we describe the current state of the world?
- How do we infer from our percepts, knowledge of unseen parts of the world?
- How does the world change as time passes?
- How does the world stay the same as time passes? (The *frame problem*.)
- How do we know the effects of our actions? (The *qualification* and *ramification problems*.)

We'll now look at one way of answering some of these questions.

Logic for knowledge representation

FOL (arguably?) seems to provide a good way in which to represent the required kinds of knowledge:

- It is *expressive*—anything you can program can be expressed.
- It is *concise*.
- It is *unambiguous*
- It can be adapted to *different contexts*.
- It has an *inference procedure*, although a semidecidable one.

In addition is has a well-defined *syntax* and *semantics*.

Logic for knowledge representation

Problem: it's quite easy to talk about things like *set theory* using FOL. For example, we can easily write axioms like

$$\forall S \,.\, \forall S' \,.\, ((\forall x \,.\, (x \in S \Leftrightarrow x \in S')) \Rightarrow S = S')$$

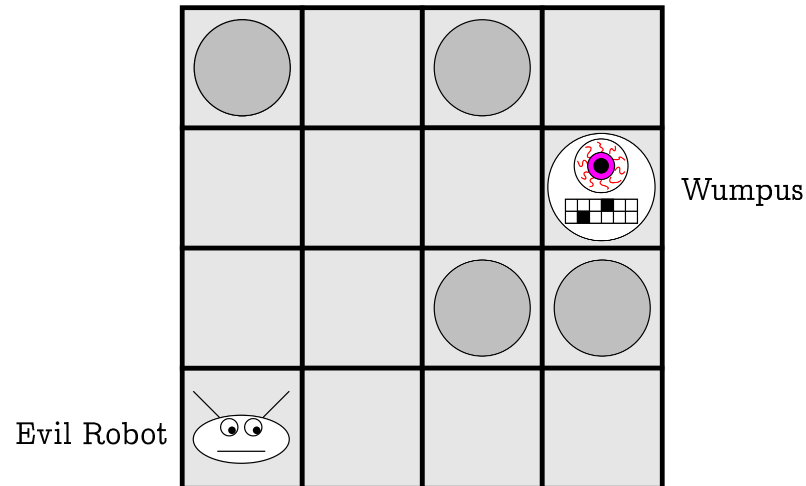
But how would we go about representing the proposition that *if you have a bucket of water and throw it at your friend they will get wet, have a bump on their head from being hit by a bucket, and the bucket will now be empty and dented*?

More importantly, how could this be represented within a wider framework for reasoning about the world?

It's time to introduce my friend, *The Wumpus*...

Wumpus world

As a simple test scenario for a knowledge-based agent we will make use of the *Wumpus World*.



The Wumpus World is a 4 by 4 grid-based cave.

EVIL ROBOT wants to enter the cave, find some gold, and get out again un-scathed.

Wumpus world

The rules of *Wumpus World*:

- Unfortunately the cave contains a number of pits, which EVIL ROBOT can fall into. Eventually his batteries will fail, and that's the end of him.
- The cave also contains the Wumpus, who is armed with state of the art *Evil Robot Obliteration Technology*.
- The Wumpus itself knows where the pits are and never falls into one.

Wumpus world

EVIL ROBOT can move around the cave at will and can perceive the following:

- In a position adjacent to the Wumpus, a stench is perceived. (Wumpuses are famed for their *lack of personal hygiene*.)
- In a position adjacent to a pit, a *breeze* is perceived.
- In the position where the gold is, a glitter is perceived.
- On trying to move into a wall, a *bump* is perceived.
- On killing the Wumpus a *scream* is perceived.

In addition, EVIL ROBOT has a single arrow, with which to try to kill the Wumpus.

"Adjacent" in the following does *not* include diagonals.

Wumpus world

So we have:

Percepts: `stench`, `breeze`, `glitter`, `bump`, `scream`.

Actions: `forward`, `turnLeft`, `turnRight`, `grab`, `release`, `shoot`, `climb`.

Of course, our aim now is *not* just to design an agent that can perform well in a single cave layout.

We want to design an agent that can *usually* perform well *regardless* of the layout of the cave.

Some nomenclature

The choice of knowledge representation language tends to lead to two important commitments:

- *Ontological commitments*: what does the world consist of?
- *Epistemological commitments*: what are the allowable states of knowledge?

Propositional logic is useful for introducing some fundamental ideas, but its ontological commitment—that the world consists of facts—sometimes makes it too limited for further use.

FOL has a different ontological commitment—the world consists of *facts*, *objects* and *relations*.

Logic for knowledge representation

The fundamental aim is to construct a *knowledge base* `KB` containing a *collection of statements* about the world—expressed in FOL—such that *useful things can be derived* from it.

Our central aim is to generate sentences that are *true*, if *the sentences in the* `KB` *are true*.

This process is based on concepts familiar from your introductory logic courses:

- Entailment: $\texttt{KB} \models \alpha$ means that the `KB` entails α.
- Proof: $\texttt{KB} \vdash_{i} \alpha$ means that α is derived from the `KB` using i. If i is *sound* then we have a *proof*.
- i is *sound* if it can generate only entailed α.
- i is *complete* if it can find a proof for *any* entailed α.

Example: Prolog

You have by now learned a little about programming in *Prolog*. For example:

```
concat([],L,L).
concat([H|T],L,[H|L2]) :- concat(T,L,L2).
```

is a program to concatenate two lists. The query

```
concat([1,2,3],[4,5],X).
```

results in

```
X = [1, 2, 3, 4, 5].
```

What's happening here? Well, Prolog is just a *more limited form of FOL* so...

Example: Prolog

... we are in fact doing inference from a `KB`:

- The Prolog programme itself is the `KB`. It expresses some *knowledge about lists*.
- The query is expressed in such a way as to *derive some new knowledge*.

How does this relate to full FOL? First of all the list notation is nothing but *syntactic sugar*. It can be removed: we define a constant called `empty` and a function called `cons`.

Now `[1,2,3]` just means `cons(1, cons(2, cons(3, empty))))` which is a term in FOL.

I will assume the use of the syntactic sugar for lists from now on.

Prolog and FOL

The program when expressed in FOL, says

$$\forall x \,.\, \mathtt{concat}(\mathtt{empty}, x, x) \wedge$$
$$\forall h, t, l_1, l_2 \,.\, \mathtt{concat}(t, l_1, l_2) \Longrightarrow \mathtt{concat}(\mathtt{cons}(h, t), l_1, \mathtt{cons}(h, l_2))$$

The rule is simple—given a Prolog program:

- *Universally quantify all the unbound variables in each line of the program* and ...
- ... *form the conjunction of the results*.

If the universally quantified lines are $L_1, L_2, \ldots, L_n$ then the Prolog programme corresponds to the KB

$$\mathtt{KB} = L_1 \wedge L_2 \wedge \cdots \wedge L_n$$

Now, what does the query mean?

Prolog and FOL

When you give the query

```
concat([1,2,3],[4,5],X).
```

to Prolog it responds by *trying to prove* the following statement

$$\texttt{KB} \Longrightarrow \exists x\, .\, \texttt{concat}([1,2,3],[4,5],x)$$

So: it tries to prove that the KB *implies the query*, and variables in the query are existentially quantified.

When a proof is found, it supplies a *value for* x that *makes the inference true*.

Prolog and FOL

Prolog differs from FOL in that, amongst other things:

- It restricts you to using *Horn clauses*.
- Its inference procedure is not a *full-blown proof procedure*.
- It does not deal with *negation* correctly.

However *the central idea also works for full-blown theorem provers*.

If you want to experiment, you can obtain *Prover9* from

`http://www.cs.unm.edu/~mccune/mace4/`

We'll see a brief example now, and a more extensive example of its use later, time permitting...

Prolog and FOL

Expressed in Prover9, the above Prolog program and query look like this:

```
set(prolog_style_variables).

% This is the translated Prolog program for list concatenation.
% Prover9 has its own syntactic sugar for lists.

formulas(assumptions).
  concat([], L, L).
  concat(T, L, L2) -> concat([H:T], L, [H:L2]).
end_of_list.

% This is the query.

formulas(goals).
  exists X concat([1, 2, 3], [4, 5], X).
end_of_list.
```

Note: it is assumed that *unbound variables are universally quantified*.

Prolog and FOL

You can try to infer a proof using

```
prover9 -f file.in
```

and the result is (in addition to a lot of other information):

```
1 concat(T,L,L2) -> concat([H:T],L,[H:L2]) # label(non_clause).  [assumption].
2 (exists X concat([1,2,3],[4,5],X)) # label(non_clause) # label(goal).  [goal].
3 concat([],A,A).  [assumption].
4 -concat(A,B,C) | concat([D:A],B,[D:C]).  [clausify(1)].
5 -concat([1,2,3],[4,5],A).  [deny(2)].
6 concat([A],B,[A:B]).  [ur(4,a,3,a)].
7 -concat([2,3],[4,5],A).  [resolve(5,a,4,b)].
8 concat([A,B],C,[A,B:C]).  [ur(4,a,6,a)].
9 $F.  [resolve(8,a,7,a)].
```

This shows that a proof is found but doesn't explicitly give a value for `X`—we'll see how to extract that later...

The fundamental idea

So the *basic idea* is: build a `KB` that encodes *knowledge about the world*, the *effects of actions* and so on.

The `KB` is a conjunction of pieces of knowledge, such that:

- A query regarding what our agent should do *can be posed in the form*

  $$\exists \texttt{actionList}.\texttt{Goal}(\ldots\ \texttt{actionList}\ \ldots)$$

- Proving that

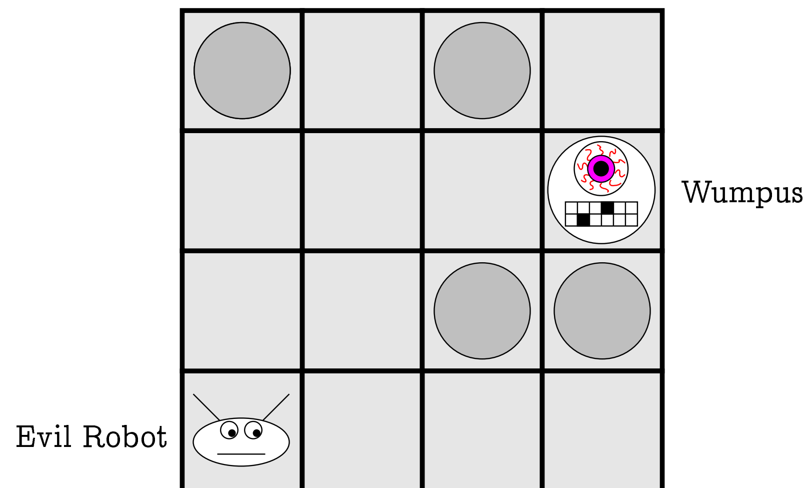
  $$\texttt{KB} \Longrightarrow \exists \texttt{actionList}.\texttt{Goal}(\ldots\ \texttt{actionList}\ \ldots)$$

  instantiates `actionList` to an *actual list of actions* that will achieve a goal represented by the `Goal` predicate.

We sometimes use the notation `ask` and `tell` to refer to *querying* and *adding to the* `KB`.

Using FOL in AI: the triumphant return of the Wumpus

We want to be able to *speculate* about the past and about *possible futures*. So:



- We include *situations* in the logical language used by our KB.
- We include *axioms* in our KB that relate to situations.

This gives rise to *situation calculus*.

Situation calculus

In *situation calculus*:

- The world consists of sequences of *situations*.
- Over time, an agent moves from one situation to another.
- Situations are changed as a result of *actions*.

In Wumpus World the actions are: `forward`, `shoot`, `grab`, `climb`, `release`, `turnRight`, `turnLeft`.

- A *situation argument* is added to items that can change over time. For example

  $$\text{At}(\text{location}, s)$$

  Items that can change over time are called *fluents*.
- A situation argument is not needed for things that don't change. These are sometimes referred to as *eternal* or *atemporal*.

Representing change as a result of actions

Situation calculus uses a function

$$\text{result}(\texttt{action}, s)$$

to denote the *new* situation arising as a result of performing the specified action in the specified situation.

$$\begin{aligned}
\text{result}(\texttt{grab}, s_0) &= s_1 \\
\text{result}(\texttt{turnLeft}, s_1) &= s_2 \\
\text{result}(\texttt{shoot}, s_2) &= s_3 \\
\text{result}(\texttt{forward}, s_3) &= s_4 \\
&\vdots
\end{aligned}$$

Axioms I: possibility axioms

The first kind of axiom we need in a KB specifies *when particular actions are possible*.

We introduce a predicate

$$\text{Poss}(\texttt{action}, s)$$

denoting that an action can be performed in situation s.

We then need a *possibility axiom* for each action. For example:

$$\text{At}(\mathtt{l}, s) \wedge \text{Available}(\texttt{gold}, \mathtt{l}, s) \Longrightarrow \text{Poss}(\texttt{grab}, s)$$

Remember that *unbound variables are universally quantified*.

Axioms II: effect axioms

Given that an action results in a new situation, we can introduce *effect axioms* to specify the properties of the new situation.

For example, to keep track of whether EVIL ROBOT has the gold we need *effect axioms* to describe the effect of picking it up:

$$\text{Poss}(\texttt{grab}, s) \Longrightarrow \text{Have}(\texttt{gold}, \text{result}(\texttt{grab}, s))$$

Effect axioms describe the way in which the world *changes*.

We would probably also include

$$\neg\text{Have}(\texttt{gold}, s_0)$$

in the KB, where s_0 is the *starting state*.

Important: we are describing *what is true* in the *situation that results* from *performing an action* in a *given situation*.

Axioms III: frame axioms

We need *frame axioms* to describe *the way in which the world stays the same*.

Example:

$$\begin{aligned}
&\mathrm{Have}(o, s) \wedge \\
&\qquad \neg(a = \texttt{release} \wedge o = \texttt{gold}) \;\wedge\; \neg(a = \texttt{shoot} \wedge o = \texttt{arrow}) \\
&\qquad \implies \mathrm{Have}(o, \mathrm{result}(a, s))
\end{aligned}$$

describes the effect of *having something and not discarding it*.

In a more general setting such an axiom might well look different. For example

$$\begin{aligned}
&\neg\mathrm{Have}(o, s) \wedge \\
&\qquad (a \neq \mathrm{grab}(o) \;\vee\; \neg(\mathrm{Available}(o, s) \;\wedge\; \mathrm{Portable}(o))) \\
&\qquad \implies \neg\mathrm{Have}(o, \mathrm{result}(a, s))
\end{aligned}$$

describes the effect of *not having something and not picking it up*.

The frame problem

The *frame problem* has historically been a major issue.

Representational frame problem: a large number of frame axioms are required to represent the many things in the world which will not change as the result of an action.

We will see how to solve this in a moment.

Inferential frame problem: when reasoning about a sequence of situations, all the unchanged properties still need to be carried through all the steps.

This can be alleviated using *planning systems* that allow us to reason efficiently when actions change only a small part of the world. There are also other remedies, which we will not cover.

Successor-state axioms

Effect axioms and frame axioms can be combined into *successor-state axioms*.

One is needed for each predicate that can change over time.

Action a is possible $\Longrightarrow$
(true in new situation $\Longleftrightarrow$
(you did something to make it true $\vee$
it was already true and you didn't make it false))

For example

$$\begin{aligned}
&\text{Poss}(a, s) \Longrightarrow \\
&\quad (\text{Have}(o, \text{result}(a, s)) \Longleftrightarrow ((a = \texttt{grab} \wedge \text{Available}(o,s)) \vee \\
&\quad (\text{Have}(o, s) \wedge \neg(a = \texttt{release} \wedge o = \texttt{gold}) \wedge \\
&\quad \neg(a = \texttt{shoot} \wedge o = \texttt{arrow}))))
\end{aligned}$$

Knowing where you are

If s_0 is the initial situation we know that

$$\mathsf{At}((1,1), s_0)$$

I am *assuming* that we've added axioms allowing us to deal with the numbers 0 to 5 and pairs of such numbers. *(Exercise: do this.)*

We need to keep track of what way we're facing. Say north is 0, south is 2, east is 1 and west is 3.

$$\mathsf{facing}(s_0) = 0$$

We need to know how motion affects location

$$\mathsf{forwardResult}((x, y), \mathtt{north}) = (x, y + 1)$$
$$\mathsf{forwardResult}((x, y), \mathtt{east}) = (x + 1, y)$$
$$\vdots$$

and

$$\mathsf{At}(l, s) \Longrightarrow \mathsf{goForward}(s) = \mathsf{forwardResult}(l, \mathsf{facing}(s))$$

Knowing where you are

The concept of adjacency is very important in the Wumpus world

$$\text{Adjacent}(l_1, l_2) \iff \exists d \; \text{forwardResult}(l_1, d) = l_2$$

We also know that the cave is 4 by 4 and surrounded by walls

$$\text{WallHere}((x, y)) \iff (x = 0 \vee y = 0 \vee x = 5 \vee y = 5)$$

It is only possible to change location by moving, and this only works if you're not facing a wall. So...

...we need a successor-state axiom:

$$\begin{aligned}
&\text{Poss}(a, s) \Longrightarrow \\
&\quad \text{At}(l, \text{result}(a, s)) \iff (l = \text{goForward}(s) \\
&\qquad \wedge\, a = \texttt{forward} \\
&\qquad \wedge \neg \text{WallHere}(l)) \\
&\qquad \vee (\text{At}(l, s) \wedge a \neq \texttt{forward})
\end{aligned}$$

Knowing where you are

It is only possible to change orientation by turning. Again, we need a successor-state axiom

$$\begin{aligned}
&\mathrm{Poss}(a, s) \Longrightarrow \\
&\quad \mathrm{facing}(\mathrm{result}(a, s)) = d \iff \\
&\qquad (a = \texttt{turnRight} \wedge d = \mathrm{mod}(\mathrm{facing}(s) + 1, 4)) \\
&\qquad \vee (a = \texttt{turnLeft} \wedge d = \mathrm{mod}(\mathrm{facing}(s) - 1, 4)) \\
&\qquad \vee (\mathrm{facing}(s) = d \wedge a \neq \texttt{turnRight} \wedge a \neq \texttt{turnLeft})
\end{aligned}$$

and so on...

The qualification and ramification problems

Qualification problem: we are in general never completely certain what conditions are required for an action to be effective.

Consider for example turning the key to start your car.

This will lead to problems if important conditions are omitted from axioms.

Ramification problem: actions tend to have implicit consequences that are large in number.

For example, if I pick up a sandwich in a dodgy sandwich shop, I will also be picking up all the bugs that live in it. I don't want to model this explicitly.

Solving the ramification problem

The ramification problem can be solved by *modifying successor-state axioms*.

For example:

$$
\begin{aligned}
&\text{Poss}(a, s) \implies \\
&\quad (\text{At}(o, l, \text{result}(a, s)) \iff \\
&\qquad (a = \text{go}(l', l) \wedge \\
&\qquad\qquad [o = \text{robot} \vee \text{Has}(\text{robot}, o, s)]) \vee \\
&\qquad (\text{At}(o, l, s) \wedge \\
&\qquad\qquad [\neg \exists l'' \,.\, a = \text{go}(l, l'') \wedge l \neq l'' \wedge \\
&\qquad\qquad \{o = \text{robot} \vee \text{Has}(\text{robot}, o, s)\}]))
\end{aligned}
$$

describes the fact that anything EVIL ROBOT is carrying moves around with him.

Deducing properties of the world: causal rules

If you know where you are, then you can think about *places* rather than just *situations*.

Synchronic rules relate properties shared by a single state of the world.

There are two kinds: *causal* and *diagnostic*.

Causal rules: some properties of the world will produce percepts.

$$\text{WumpusAt}(l_1) \wedge \text{Adjacent}(l_1, l_2) \Longrightarrow \text{StenchAt}(l_2)$$

$$\text{PitAt}(l_1) \wedge \text{Adjacent}(l_1, l_2) \Longrightarrow \text{BreezeAt}(l_2)$$

Systems reasoning with such rules are known as *model-based* reasoning systems.

Deducing properties of the world: diagnostic rules

Diagnostic rules: infer properties of the world from percepts.

For example:

$$\mathrm{At}(l, s) \land \mathrm{Breeze}(s) \implies \mathrm{BreezeAt}(l)$$

$$\mathrm{At}(l, s) \land \mathtt{Stench}(s) \implies \mathtt{StenchAt}(l)$$

These may not be very strong.

The difference between model-based and diagnostic reasoning can be important. For example, medical diagnosis can be done based on symptoms or based on a model of disease.

General axioms for situations and objects

Note: in FOL, if we have two constants `robot` and `gold` then an interpretation is free to assign them to be the same thing.

This is not something we want to allow.

Unique names axioms state that each pair of distinct items in our model of the world must be different

$$
\begin{aligned}
\texttt{robot} &\neq \texttt{gold} \\
\texttt{robot} &\neq \texttt{arrow} \\
\texttt{robot} &\neq \texttt{wumpus} \\
&\vdots \\
\texttt{wumpus} &\neq \texttt{gold} \\
&\vdots
\end{aligned}
$$

General axioms for situations and objects

Unique actions axioms state that actions must share this property, so for each pair of actions

$$\begin{aligned} \mathtt{go}(l, l') &\neq \mathtt{grab} \\ \mathtt{go}(l, l') &\neq \mathtt{drop}(o) \\ &\vdots \\ \mathtt{drop}(o) &\neq \mathtt{shoot} \\ &\vdots \end{aligned}$$

and in addition we need to define equality for actions, so for each action

$$\begin{aligned} \mathtt{go}(l, l') = \mathtt{go}(l'', l''') &\iff l = l'' \wedge l' = l''' \\ \mathtt{drop}(o) = \mathtt{drop}(o') &\iff o = o' \\ &\vdots \end{aligned}$$

General axioms for situations and objects

The situations are *ordered* so

$$s_0 \neq \text{result}(a, s)$$

and situations are *distinct* so

$$\text{result}(a, s) = \text{result}(a', s') \iff a = a' \wedge s = s'$$

Strictly speaking we should be using a *many-sorted* version of FOL.

In such a system variables can be divided into *sorts* which are implicitly separate from one another.

The start state

Finally, we're going to need to specify *what's true in the start state*.

For example

$$\begin{aligned} \text{At}(\texttt{robot}, [1,1], s_0) \\ \text{At}(\texttt{wumpus}, [3,4], s_0) \\ \text{Has}(\texttt{robot}, \texttt{arrow}, s_0) \\ \vdots \end{aligned}$$

and so on.

Sequences of situations

We know that the function result tells us about the situation resulting from performing an action in an earlier situation.

How can this help us find *sequences of actions to get things done*?

Define

$$\begin{aligned}\text{Sequence}([\,], s, s') &= s' = s \\ \text{Sequence}([a], s, s') &= \text{Poss}(a, s) \wedge s' = \text{result}(a, s) \\ \text{Sequence}(a :: as, s, s') &= \exists t \;.\; \text{Sequence}([a], s, t) \wedge \text{Sequence}(as, t, s')\end{aligned}$$

To obtain a *sequence of actions that achieves* Goal(s) we can use the query

$$\exists a \; \exists s \;.\; \text{Sequence}(a, s_0, s) \wedge \text{Goal}(s)$$

Problems

1. There have in fact been *two* queries suggested in the notes for obtaining a sequence of actions. The details for

$$\exists a \; \exists s \; . \; \text{Sequence}(a, s_0, s) \wedge \text{Goal}(s)$$

   were given on the last slide, but earlier in the notes the format

$$\exists \texttt{actionList.Goal(... actionList ...)}$$

   was suggested. Explain how this alternative form of query might be made to work.

2. Making correct use of the situation calculus, write the sentences in FOL required to implement the following actions in Wumpus World:

   - `Climb`
   - `Shoot`

Problems

Paper 9, Question 8, 2003 - part 2

You wish to construct a robotic pet cat for the purposes of entertainment. One purpose of the cat is to scratch valuable objects when the owner is not present. Give a brief general description of *situation calculus* and describe how it might be used for knowledge representation by the robot. Include in your answer one example each of a *frame axiom*, an *effect axiom*, and a *successor-state axiom*, along with example definitions of suitable predicates and functions. [12 marks]

Knowledge representation and reasoning

It should be clear that generating sequences of actions by inference in FOL is highly non-trivial.

Ideally we'd like to maintain an *expressive* language while *restricting* it enough to be able to do inference *efficiently*.

Further aims:

- To give a brief introduction to *semantic networks* and *frames* for knowledge representation.
- To see how *inheritance* can be applied as a reasoning method.
- To look at the use of *rules* for knowledge representation, along with *forward chaining* and *backward chaining* for reasoning.

Further reading: *The Essence of Artificial Intelligence*, Alison Cawsey. Prentice Hall, 1998.

Frames and semantic networks

Frames and semantic networks represent knowledge in the form of *classes of objects* and *relationships between them*:

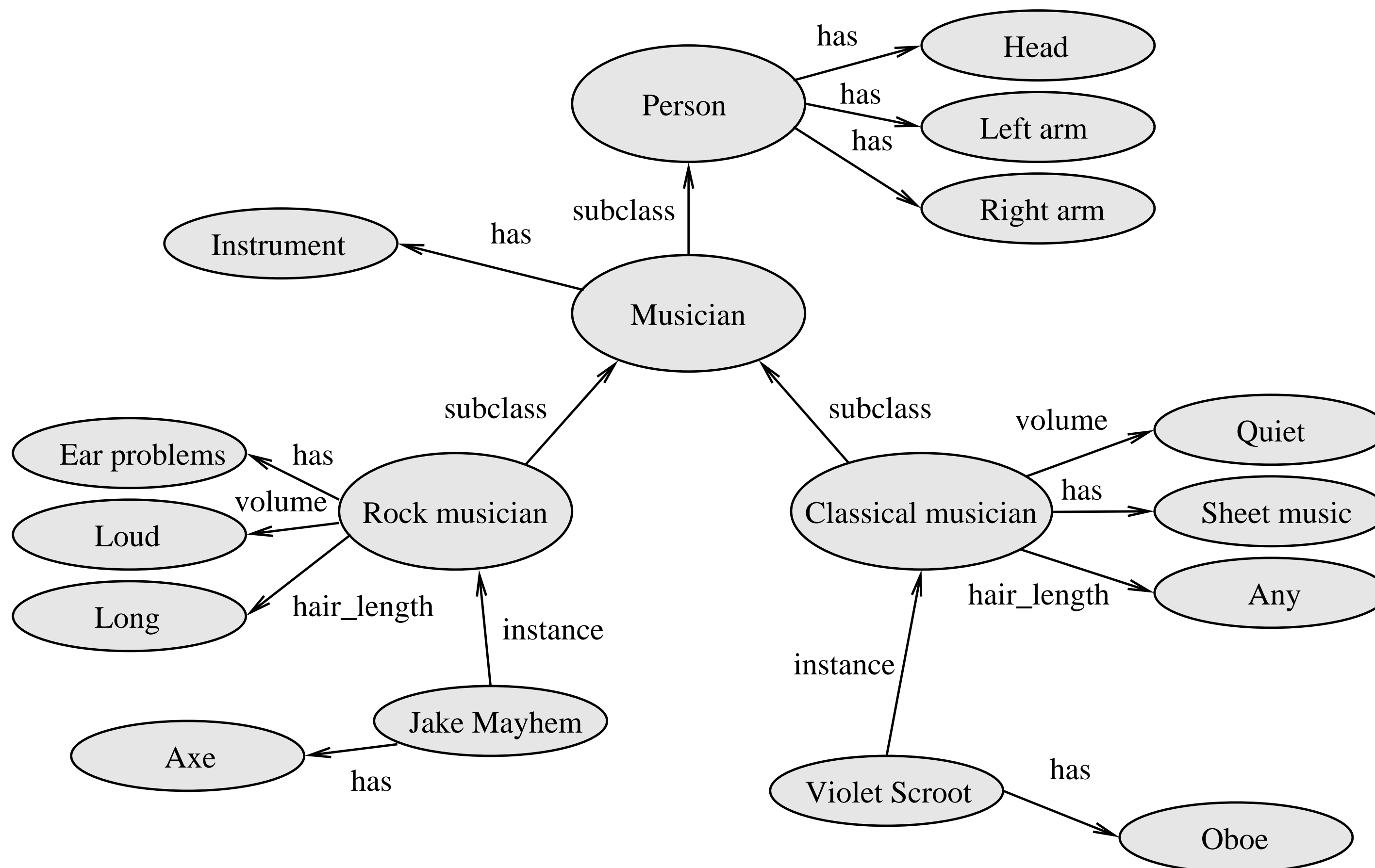
- The *subclass* and *instance* relationships are emphasised.
- We form *class hierarchies* in which *inheritance* is supported and provides the main *inference mechanism*.

As a result inference is quite limited.

We also need to be extremely careful about *semantics*.

The only major difference between the two ideas is *notational*.

Example of a semantic network



Frames

Frames once again support inheritance through the *subclass relationship*.

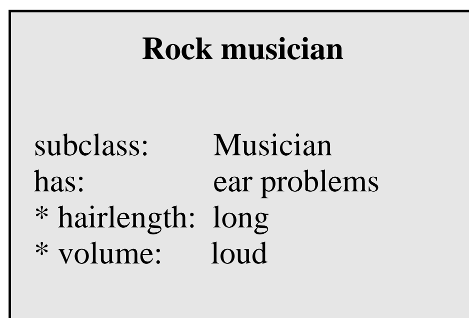
`has`, `hairlength`, `volume` *etc* are *slots*.

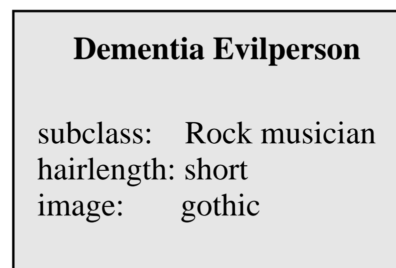
`long`, `loud`, `instrument` *etc* are *slot values*.

These are a direct predecessor of *object-oriented programming languages*.

Defaults

Both approaches to knowledge representation are able to incorporate *defaults*:

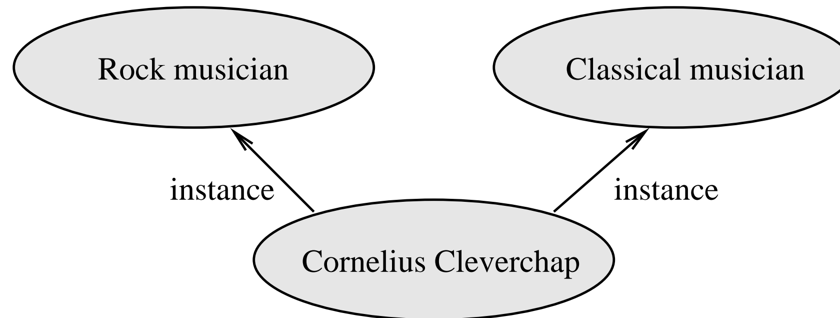




Starred slots are *typical values* associated with subclasses and instances, but *can be overridden*.

Multiple inheritance

Both approaches can incorporate *multiple inheritance*, at a cost:



- What is `hairlength` for `Cornelius` if we're trying to use inheritance to establish it?
- This can be overcome initially by specifying which class is inherited from *in preference* when there's a conflict.
- But the problem is still not entirely solved—what if we want to prefer inheritance of some things from one class, but inheritance of others from a different one?

Other issues

- Slots and slot values can themselves be frames. For example `Dementia` may have an instrument slot with the value `Electric harp`, which itself may have properties described in a frame.
- Slots can have *specified attributes*. For example, we might specify that `instrument` can have multiple values, that each value can only be an instance of `Instrument`, that each value has a slot called `owned_by` and so on.
- Slots may contain arbitrary pieces of program. This is known as *procedural attachment*. The fragment might be executed to return the slot's value, or update the values in other slots *etc*.

Rule-based systems

A rule-based system requires three things:

1. A set of *if–then rules*. These denote specific pieces of knowledge about the world.

   They should be interpreted similarly to logical implication.

   Such rules denote *what to do* or *what can be inferred* under given circumstances.

2. A collection of *facts* denoting what the system regards as currently true about the world.

3. An interpreter able to apply the current rules in the light of the current facts.

Forward chaining

The first of two basic kinds of interpreter *begins with established facts and then applies rules to them*.

This is a *data-driven* process. It is appropriate if we know the *initial facts* but not the required conclusion.

Example: XCON—used for configuring VAX computers.

In addition:

- We maintain a *working memory*, typically of what has been inferred so far.
- Rules are often *condition-action rules*, where the right-hand side specifies an action such as adding or removing something from working memory, printing a message *etc*.
- In some cases actions might be entire program fragments.

Forward chaining

The basic algorithm is:

1. Find all the rules that can fire, based on the current working memory.
2. Select a rule to fire. This requires a *conflict resolution strategy*.
3. Carry out the action specified, possibly updating the working memory.

Repeat this process until either *no rules can be used* or a *halt* appears in the working memory.

Example

Condition–action rules

```
dry_mouth -> ADD thirsty
thirsty -> ADD get_drink
get_drink AND no_work -> ADD go_bar
working -> ADD no_work
no_work -> DELETE working
```

Working memory

```
dry_mouth
working
```

Interpreter

Example

Progress is as follows:

1. The rule

$$\texttt{dry_mouth} \Longrightarrow \text{ADD } \texttt{thirsty}$$

   fires adding `thirsty` to working memory.

2. The rule

$$\texttt{thirsty} \Longrightarrow \text{ADD } \texttt{get_drink}$$

   fires adding `get_drink` to working memory.

3. The rule

$$\texttt{working} \Longrightarrow \text{ADD } \texttt{no_work}$$

   fires adding `no_work` to working memory.

4. The rule

$$\texttt{get_drink} \text{ AND } \texttt{no_work} \Longrightarrow \text{ADD } \texttt{go_bar}$$

   fires, and we establish that it's time to go to the bar.

Conflict resolution

Clearly in any more realistic system we expect to have to deal with a scenario where *two or more rules can be fired at any one time*:

- Which rule we choose can clearly affect the outcome.
- We might also want to attempt to avoid inferring an abundance of useless information.

We therefore need a means of *resolving such conflicts*.

Conflict resolution

Common *conflict resolution strategies* are:

- Prefer rules involving more recently added facts.
- Prefer rules that are *more specific*. For example

  $$\texttt{patient_coughing} \Longrightarrow \text{ADD } \texttt{lung_problem}$$

  is more general than

  $\texttt{patient_coughing}$ AND $\texttt{patient_smoker} \Longrightarrow$ ADD $\texttt{lung_cancer}$.

  This allows us to define exceptions to general rules.
- Allow the designer of the rules to specify priorities.
- Fire all rules *simultaneously*—this essentially involves following all chains of inference at once.

Reason maintenance

Some systems will allow information to be removed from the working memory if it is no longer *justified*.

For example, we might find that

`patient_coughing`

and

`patient_smoker`

are in working memory, and hence fire

`patient_coughing` AND `patient_smoker` $\Longrightarrow$ ADD `lung_cancer`

but later infer something that causes `patient_coughing` to be *withdrawn* from working memory.

The justification for `lung_cancer` has been removed, and so it should perhaps be removed also.

Pattern matching

In general rules may be expressed in a slightly more flexible form involving *variables* which can work in conjunction with *pattern matching*.

For example the rule

$$\texttt{coughs}(X) \text{ AND } \texttt{smoker}(X) \Longrightarrow \text{ADD } \texttt{lung_cancer}(X)$$

contains the variable X.

If the working memory contains coughs(neddy) and smoker(neddy) then

$$X = \texttt{neddy}$$

provides a match and

$$\texttt{lung_cancer}(\texttt{neddy})$$

is added to the working memory.

Backward chaining

The second basic kind of interpreter begins with a *goal* and finds a rule that would achieve it.

It then works *backwards*, trying to achieve the resulting earlier goals in the succession of inferences.

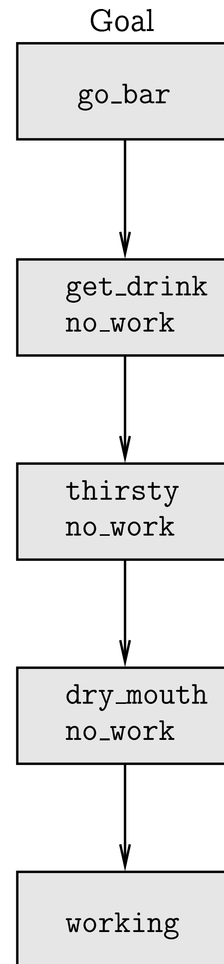
Example: MYCIN—medical diagnosis with a small number of conditions.

This is a *goal-driven* process. If you want to *test a hypothesis* or you have some idea of a likely conclusion it can be more efficient than forward chaining.

Example

Working memory

dry_mouth
working



To establish go_bar we have to establish get_drink and no_work. These are the new goals.

Try first to establish get_drink. This can be done by establishing thirsty.

thirsty can be established by establishing dry_mouth. This is in the working memory so we're done.

Finally, we can establish no_work by establishing working. This is in the working memory so the process has finished.

Example with backtracking

If at some point more than one rule has the required conclusion then we can *backtrack*.

Example: *Prolog* backtracks, and incorporates pattern matching. It orders attempts according to the order in which rules appear in the program.

Example: having added

$$\texttt{up_early} \Longrightarrow \text{ADD } \texttt{tired}$$

and

$$\texttt{tired} \text{ AND } \texttt{lazy} \Longrightarrow \text{ADD } \texttt{go_bar}$$

to the rules, and `up_early` to the working memory:

Example with backtracking

Working memory

dry_mouth
working
up_early

Goal

go_bar

tired
lazy

Attempt to establish go_bar by establishing tired and lazy.

get_drink
no_work

up_early
lazy

This can be done by establishing up_early and lazy.
up_early is in the working memory so we're done.

thirsty
no_work

Process proceeds as before

lazy

We can not establisg lazy and so we backtrack and try a different approach.

dry_mouth
no_work

working