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.

Frames and semantic networks

Frames and semantic networks represent knowledge in the form of \textit{classes of objects} and \textit{relationships between them}:

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

As a result inference is quite limited.

We also need to be extremely careful about \textit{semantics}.

The only major difference between the two ideas is \textit{notational}. 

Example of a semantic network

Person
  has
  has
Left arm
  Right arm

Musician
  has
  has subclass
Instrument

Musician
  has
  has subclass
Rock musician
  has
  volume
  hair_length
  instance
Jake Mayhem

Musician
  has subclass
Classical musician
  has
  hair_length
  Any
  Sheet music

Musician
  has subclass
Quiet

Musical instrument

Oboe

Axe

Rock musician
  hair_length
  instance
Violet Scroot

Ear problems
  volume
Loud
Long

Violet Scroot
  has
Oboe

Volume

Sheet music

Hair length

Person
  has
Head
  Left arm
  Right arm

Axe

Rock musician
  has
Violet Scroot

The diagram illustrates a semantic network, showing relationships between different concepts and instances. It includes entities like persons, musicians, instruments, and their respective properties and relationships.
Frames

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

<table>
<thead>
<tr>
<th>Rock musician</th>
<th>Musician</th>
</tr>
</thead>
<tbody>
<tr>
<td>subclass: Musician</td>
<td>subclass: Person</td>
</tr>
<tr>
<td>has: ear problems</td>
<td>has: instrument</td>
</tr>
<tr>
<td>hairlength: long</td>
<td></td>
</tr>
<tr>
<td>volume: loud</td>
<td></td>
</tr>
</tbody>
</table>

*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*:

<table>
<thead>
<tr>
<th>Rock musician</th>
<th>Dementia Evilperson</th>
</tr>
</thead>
<tbody>
<tr>
<td>subclass:</td>
<td>Musician</td>
</tr>
<tr>
<td>has:</td>
<td>ear problems</td>
</tr>
<tr>
<td>* hairlength:</td>
<td>long</td>
</tr>
<tr>
<td>* volume:</td>
<td>loud</td>
</tr>
<tr>
<td>subclass:</td>
<td>Rock musician</td>
</tr>
<tr>
<td>hairlength:</td>
<td>short</td>
</tr>
<tr>
<td>image:</td>
<td>gothic</td>
</tr>
</tbody>
</table>

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 hair length 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 Electricarp, 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
  – Each value can only be an instance of Instrument
  – 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.
**Condition–action rules**

- dry_mouth $\rightarrow$ ADD thirsty
- thirsty $\rightarrow$ ADD get_drink
- get_drink AND no_work $\rightarrow$ ADD go_bar
- working $\rightarrow$ ADD no_work
- no_work $\rightarrow$ DELETE working

**Working memory**

- dry_mouth
- working

**Interpreter**
Example

Progress is as follows:

1. The rule
   \[ \text{dry_mouth} \rightarrow \text{ADD thirsty} \]
   fires adding \text{thirsty} to working memory.

2. The rule
   \[ \text{thirsty} \rightarrow \text{ADD get_drink} \]
   fires adding \text{get_drink} to working memory.

3. The rule
   \[ \text{working} \rightarrow \text{ADD no_work} \]
   fires adding \text{no_work} to working memory.

4. The rule
   \[ \text{get_drink AND no_work} \rightarrow \text{ADD go_bar} \]
   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*. Common *conflict resolution strategies* are:

- Prefer rules involving more recently added facts.
- Prefer rules that are *more specific*. For example
  
  \[
  \text{patient\_coughing \rightarrow ADD lung\_problem}
  \]
  
  is more general than
  
  \[
  \text{patient\_coughing \ AND \ patient\_smoker \rightarrow ADD lung\_cancer.}
  \]
- 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

\[ \text{patient_coughing} \]

and

\[ \text{patient_smoker} \]

are in working memory, and hence fire

\[ \text{patient_coughing AND patient_smoker} \rightarrow \text{ADD lung_cancer} \]

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

\[ \text{coughs}(X) \text{ AND } \text{smoker}(X) \rightarrow \text{ADD } \text{lung_cancer}(X) \]

contains the variable \( X \).

If the working memory contains \( \text{coughs}(\text{neddy}) \) and \( \text{smoker}(\text{neddy}) \) then

\[ X = \text{neddy} \]

provides a match and

\[ \text{lung_cancer}(\text{neddy}) \]

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.
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

\[
\text{up\_early} \rightarrow \text{ADD tired}
\]

and

\[
\text{tired AND lazy} \rightarrow \text{ADD go\_bar}
\]

to the rules, and *up\_early* to the working memory:
Example with backtracking

Working memory
- dry_mouth
- working
- up_early

Goal
- go_bar

Attempt to establish go_bar by establishing tired and lazy.

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

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

Process proceeds as before

- thirsty
- no_work

- dry_mouth
- no_work

- working