Today

- Predicate Completion
- Reasoning Maintenance

Recall Circumscription Axiom

\[ \forall P' \quad T(P') \land \forall x \ (P'(x) \rightarrow P(x)) \]
\[ \rightarrow \forall x \ (P'(x) \leftrightarrow P(x)) \]

Here \( T(P') \) is \( T \) with each occurrence of \( P \) replaced with \( P' \).

Predicate Completion

Suppose that a KB has a single formula \( \text{foo}(a) \) – this formula is equivalent to \( \forall x \ (x = a \rightarrow \text{foo}(x)) \).

This second form looks like one half of a definition. To complete the predicate, we add the other half of the definition to the KB, namely \( \forall x \ \text{foo}(x) \rightarrow x = a \) to form an extended KB.

We now describe a procedure to calculate the completion of a KB in a certain form.

Solitary predicate

Say a set of clauses is solitary in \( p \) if each clause with a positive occurrence of \( p \) has only one occurrence of \( p \).

A predicate in a clause is positive if it occurs in the scope of an even number of negations (or none at all); appearing in the antecedent of an implication counts as being in the scope of a negation. If the clause is written using a single implication, the occurrence is positive iff it follows the implication.
Solitary ctd

For example, the clause
\[ q(x) \land r(x, a) \rightarrow p(a) \]
is solitary in \( p \), but the following is not:
\[ p(b) \rightarrow (q(a) \lor q(b) \lor p(a)) \].
The completion procedure works for clauses solitary in \( p \).
We write \( \text{Comp}(\text{KB}, \text{foo}) \) for the completion for the for the predicate \( \text{foo} \).

Completion procedure

\begin{itemize}
\item Write clauses solitary in \( p \) in the form
\[ \forall y (Q_1 \land \ldots \land Q_n) \rightarrow p(t) \]
where \( t \) may be a tuple of terms, and \( y \) a tuple of variables appearing in \( Q_1 \land \ldots \land Q_n \).
\item Put this in the equivalent form
\[ \forall y \forall x ((x = t \land Q_1 \land \ldots \land Q_n) \rightarrow p(x)) \].
\item Put this in the equivalent form
\[ \forall x \exists y (x = t \land Q_1 \land \ldots \land Q_n) \rightarrow p(x) \].
\end{itemize}

Completion ctd

This last equivalence follows since \( \forall y (f(y) \rightarrow g) \) and \( (\exists y f(y)) \rightarrow g \) are equivalent (if \( y \) does not occur in \( g \)).

\begin{itemize}
\item Do the same for each clause solitary in \( p \). If the first clause is now in the form
\[ \forall x E_1 \rightarrow p(x) \],
this gives a number of clauses
\[ \forall x E_2 \rightarrow p(x) \]
\[ \vdots \]
\[ \forall x E_n \rightarrow p(x) \]
which can be combined to give
\[ \forall x ((E_1 \lor E_2 \lor \ldots \lor E_n) \rightarrow p(x)) \]
\end{itemize}

Completion ctd

So far we have something equivalent to the original KB. Now we add the completion formula:
\[ \forall x (p(x) \rightarrow (E_1 \lor E_2 \lor \ldots \lor E_n)) \].
Example

Take the KB

\[ \forall x \ (\text{scottish}(x) \rightarrow \text{british}(x)) \]

\[ \text{british}(\text{fred}) \]

\[ \neg \text{scottish}(\text{mary}). \]

We want to find the completion for the predicate \text{british}. After rewriting, we get

\[ \forall x \ ((\text{scottish}(x) \lor x = \text{fred}) \rightarrow \text{british}(x)). \]

\((\text{KB}, \text{british})\) is got by adding the reverse implication to KB, giving the new formula:

\[ \forall x \ (\text{british}(x) \rightarrow (\text{scottish}(x) \lor x = \text{fred}). \]

When take completion?

It is sensible to take the completion for a predicate \( p \) if we think that the only way that \( p \) can hold is if it already holds from the original KB.

The process is non-monotonic, since we can have

\( (\text{KB}, p) \vdash F \) and

\( \neg (\text{Comp}(\text{KB} \cup \{ G \}, p) \vdash F) \)

It is known that if KB is solitary in \( p \) and KB is consistent, then \( \text{Comp}(\text{KB}, p) \) will also be consistent.

Circumscription and Predicate Completion

Taking the predicate completion for a solitary predicate is in fact equivalent to taking the circumscription of the KB for that predicate.

We will not give the details of this. Notice that this gives a simple way to compute circumscription under these circumstances.

Integrating new information

We have already looked at the problem of keeping track of a dynamic (i.e. changing) KB, when new information is a candidate for being added to the KB. Although we talk about “Knowledge Bases”, in practice there are many assumptions and guesses included in a KB, many of which might be wrong, casting doubt on conclusions we get from the KB. If we are using a non-monotonic reasoning engine (e.g. the CWA is involved), then adding new information can invalidate earlier information, even though the KB remains consistent.

How can we keep track of these dependencies?
A quick fix

We can tag any deduced statement with a time tag (e.g. a sequence number). If a statement deduced at time $T$ is invalidated, for whatever reason, then throw it away, and all subsequent reasoning: go back to time $T - 1$ and redo as much as goes through of the previous reasoning.

This is wasteful (but may be fine depending on the speed of the machines, the size of the KB, and the complexity of the reasoning required).

**Example**

Time T: add "earth is flat"
Time T+1: add "Edinburgh is the capital of Scotland"
Time T+2: contradict "earth is flat"

So throw away second statement unnecessarily!

A better approach

In a truth (or reasoning) system (TMS/RMS):

- Dependencies are represented by a set of nodes with justifications;
- Each node is labelled with a statement in, or deduced from, the KB;
- Each node has zero or more justifications, i.e. records of how the statement came to be believed.

There should be only one node for any given statement; thus we build a system of nodes and interlocking justifications.

We can mark some initial statements as facts and others as assumptions – just means that we look among latter for problem cases.

Two versions of TMS

Doyle’s TMS

Suppose our reasoning engine (separate from the TMS) signals that the current information is contradictory (e.g. logically inconsistent). Then something must change in the KB.

- The TMS gives us a choice of which bits of the network might be to blame.
- TMS (or user) picks one of the possibilities.
- TMS updates the network by removing justifications — but keeps a record to allow these to be put back if the blame is shifted later on.

de Kleer’s version

Here the TMS works harder

- build up all possible justification patterns by forward chaining
- record for a node all the possible justifications
- when there is inconsistency, get choice of assumptions to blame, as before
- allows efficient propagation of new valid justifications.
de Kleer’s TMS

de Kleer aims to record the reasoning dependencies, rather than the IN/OUT of Doyle. Instead, the system records only how deductions would depend on any assumptions. That is, the job of ATMS is to record under what sets of assumptions any conclusion holds. In ATMS, each node has, in addition to the tag showing what it stands for (what you would normally call its label), something else which de Kleer confusingly terms a label; this is a set of sets of assumptions. The node follows from any one of such sets of assumptions if every assumption in the set holds.

If a node turns out to be contradictory, according to the reasoning system, then the ATMS notes that all the sets of assumptions labelling it lead to a contradiction.

Noting contradictions

The convenient way to do this noting is to have a node which stands for ‘false’, and to add any such contradictory sets to its label. For each node, a check is made such that none of its sets of assumptions is a superset of any other; if so, it should be deleted. The ATMS does this to ensure that no superfluous assumption appears anywhere in a set within any label.

Thus, if one argument uses only a subset of another argument’s assumptions, then we forget about the more extravagant argument.

How it works

To have a look at the idea of how ATMS functions, suppose that early on in the deductive process (performed by a reasoning engine) there have been nodes for assumptions $a_1 \ldots a_5$, and that there are already two nodes A and B, with labels as follows:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{ $a_1$, $a_2$ }</td>
<td>{ $a_2$, $a_3$, $a_4$ }</td>
</tr>
<tr>
<td>2</td>
<td>{ $a_4$, $a_5$ }</td>
<td>{ $a_3$, $a_5$ }</td>
</tr>
</tbody>
</table>

Suppose the false node contains \{ $a_4$, $a_5$ \}.

Suppose we want to add a justification that A and B imply C (C a new node).

Computing new label

First, find the pairwise sets of assumptions, one set for pair from each label:

1 \{ $a_1$, $a_2$ \}
2 \{ $a_1$, $a_2$, $a_3$ \}
3 \{ $a_1$, $a_2$, $a_4$ \}
4 \{ $a_1$, $a_2$, $a_5$ \}
5 \{ $a_2$, $a_3$, $a_5$ \}
6 \{ $a_2$, $a_3$, $a_5$ \}

Now remove any that has another as a subset (here, 2,3,4).

Now remove any with contradictory assumptions as a subset (here, 6).

So get new label (blend with old label, if C already there).
**Blending labels**

Take the union of the two labels, delete any set in this union which has another member in the union as a subset of it. Since the new and old labels were already consistent there is no need to go looking for sets which have contradictory subsets, this time. If the whole process changes the label for C, and recorded justifications show that other nodes depended on C, then the label changes have to be propagated forward to those other nodes, and on from them, and so on.

**Propagating information about inconsistency**

Suppose, at a certain stage, C is found to be a nogood by the reasoning system (that is, every set of assumptions labelling it indicates a contradiction). Then each member of its label is added to the label of the specific node standing for ‘false’, and all such members, and their supersets, are removed from the label of every other node. Any set of (inconsistent) assumptions being a superset of another within the ‘false’ node’s label itself is also removed.

**Advantages**

First, all the consequences of a set of assumptions can be explored together; no backtracking is involved, and the system is not striving to maintain one mutually consistent set of assumptions as in Doyle’s TMS. This suits certain kinds of application.

Second, it can be very efficiently implemented, since the main operations involved are set operations such as union and subset checking. If sets are represented as bit strings these kinds of operations can be performed directly by hardware.

**Problems?**

The obvious disadvantage is that the system is driven by forward-chaining reasoning, so there is no natural progression towards a particular desired goal. There is considerably more to ATMS than this; we have only covered the basic idea. For example, in some applications it might be desirable to constrain the ATMS to considering only those sets of assumptions which contain at least one, or perhaps exactly one, of a given set (of particular assumptions).
Summary

• Predicate Completion

• Computing circumscription

• de Kleer’s ATMS