Today

- Clark completion
- Lloyd-Topor transformation
Reminder: Negation by failure

Prolog does not distinguish between being unable to find a derivation, and claiming that the query is false; that is, it does not distinguish between the “false” and the “unknown” values we have above.

When we take a Prolog response of no. as indicating that a query is false, we are making use of the idea of negation as failure: if a statement cannot be derived, then it is false.

Clearly, this assumption is not always valid! If some information is not present in the program, failure to find a derivation should not let us conclude that the query is false – we just don’t have the information to decide.
Closed World Assumption as an augmented $T$

We can define the effect of the CWA using the standard logic we saw earlier. Given a $T$ written in first-order logic, we augment $T$ to get a bigger set of formulas $CWA(T)$; the extra formulas we add are:

$$X_T = \{ \neg p(t_1, \ldots, t_n) : \text{not } T \vdash p(t_1, \ldots, t_n) \}$$

Now we can define what it is to follow from $T$ using CWA: a formula $Q$ follows from $T$ using the CWA iff

$$T \cup X_T \models Q$$

One use of CWA is in looking at a failed Prolog query of the form

?- property(t1,t2).

as saying that the query is in fact false.
Logic and monotonic reasoning

It’s a basic feature of standard logic that it is monotonic: if we add new assumptions to a theory, we never invalidate any conclusions we could already make.

In other words, if $Q$ follows logically from a set of statements $KB$, and $X$ is a set of statements, then $Q$ follows from $KB$ together with $X$.

If $T \models Q$, then $T \cup X \models Q$

Reasoning with the CWA does not have this property; we say it is non-monotonic. Adding extra information can invalidate earlier conclusions.
Example

From our toy example, form a new KB by adding \textit{poor}(\textit{fred}) to get the new $T'$:

\[
\text{poor}(\textit{jane}) \\
\text{poor}(\textit{jane}) \rightarrow \text{happy}(\textit{jane}) \\
\text{happy}(\textit{fred}) \\
\text{poor}(\textit{fred})
\]

Now $\neg\text{poor}(\textit{fred})$ is not in $X_{T'}$, and so we do not have $\text{CWA}[T'] \models \neg\text{poor}(\textit{fred})$ any more.
Negation by Failure and Least Herbrand Model

How does negation by failure fit with the Least Herbrand model?
Suppose we have a ground query (i.e. with no variables)

\(?- \text{p}(t,v).\)

Recall that \(\text{p}(t,v)\) is true in the Least Herbrand model \(\mathcal{M}\) if and only if it is provable by Backchain inference. If Prolog returns "no" to the query, that means that there is no Backchain derivation, and so \(\text{p}(t,v)\) is false in \(\mathcal{M}\).

So negation by failure, for ground queries, gives the correct answer according to the Least Herbrand model.

Negation by failure in general applies to goals which are not ground as well, however, and in that case is not always sound in this sense.
Clark completion

When negation by failure is used, we lose the pretty picture we had before of the relationship between:

- **logical inference** in predicate logic, from definite clauses; and

- **Derivations** in the Backchain inference system.

There is a way of starting from definite clauses $S$, and computing an extended set of predicate calculus statements $\text{Comp}(S)$, and extending the Backchain inference system with negation by failure, to recover the desired connection again.
Completion

Suppose that theory 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 theory, namely $\forall x \text{foo}(x) \rightarrow x = a$.

We now describe a procedure to calculate the completion of a set of definite clauses.
Completion procedure

• Suppose start with a definite clause

\[ \forall y \ ((Q_1 \land \ldots \land Q_n) \rightarrow p(t)) \]

where \( t \) may be a tuple of terms, and \( y \) a tuple of all the variables appearing in \( Q_1 \land \ldots \land Q_n \).

• Put this in the equivalent form

\[ \forall y \ \forall x \ ((x = t \land Q_1 \land \ldots \land Q_n) \rightarrow p(x)) \].

• Put this in the equivalent form

\[ \forall x \ \exists y \ (x = t \land Q_1 \land \ldots \land Q_n) \rightarrow p(x) \). \]
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 $g$).

• Do the same for each clause of the predicate $p$. If the first clause is now in the form $\forall x \ E_1 \rightarrow p(x)$, this gives a number of clauses

\[ \forall x \ E_1 \rightarrow p(x) \]
\[ \forall x \ E_2 \rightarrow p(x) \]
\[ \vdots \]
\[ \forall x \ E_m \rightarrow p(x) \]

which can be combined to give $\forall x \ ((E_1 \lor E_2 \lor \ldots \lor E_m) \rightarrow p(x))$
Completion ctd

• So far we have something equivalent to the original KB. Now we replace these clauses with the completion formula:

\[ \forall x \ (p(x) \leftrightarrow (E_1 \lor E_2 \lor \ldots \lor E_m)). \]
Example

Take the clauses:

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

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

To take the completion, we get first:

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

and so the completed program is given by the new formula:

\( \forall x \ (\text{british}(x) \leftrightarrow (\text{scottish}(x) \lor x = \text{fred})). \)

Note that from the completion we can deduce \( \neg \text{british}(\text{dai}) \), which does not follow logically from the initial clauses.
Clark Completion

The Clark completion works by replacing every predicate in this way.

For predicates that do not appear in the head of any clause, we add explicit negations; eg for `foo/3` add

\[ \forall x \forall y \forall z \neg \text{foo}(x, y, z) \]

This gives a standard way of thinking about logic programs. Lloyd says:

Even though a programmer only gives a logic programming system the general program, the understanding is that, conceptually, the general program is completed by the system and that the programmer is actually programming with the completion.

(Foundations of Logic Programming, p 71)
Properties of the Completion

- \( S \) follows logically from \( \text{Comp}(S) \)
  (since we built the completion as a stronger theory, by replacing implication with equivalence)

- the completion of a set of definite clauses is always consistent (it always has a model).
  (the reason for this will be a tutorial topic)

- the completion adds no \textit{positive} information: for atomic statements \( A \),

\[
S \models A \text{ if and only if } \text{Comp}(S) \models A
\]

The completion lets us conclude new negative information, though, justifying the closed world assumption.
Lloyd-Topor transformation

Here we look in the other direction.

Can we allow programs to be written in a more expressive syntax, and find a way to transform them into something closer to definite clauses?

Lloyd and Topor found a way to allow arbitrary first order formulas in the body of statements, and transform this into general programs.

A general program is one where a single negation is allowed in front of each atom in the body of a clause. So a general clause now has the shape:

\[ H :- (\text{not}) B_1, \ldots, (\text{not}) B_n. \]

We will see that straightforward execution of such a program with negation as failure is problematic, however.
Lloyd Topor rules

The transformation is given by a number of operations that can be applied to clauses. The rules are then applied again and again wherever possible until no rule applies. This transforms a program where arbitrary predicate formulas appear in the body to a general program as just defined.

Disjunction:

\[ A \leftarrow V \lor W \quad \Rightarrow \quad A \leftarrow V \text{ and } A \leftarrow W \]

\[ A \leftarrow \neg(V \lor W) \quad \Rightarrow \quad A \leftarrow \neg V, \neg W \]

Conjunction

\[ A \leftarrow V \land W \quad \Rightarrow \quad A \leftarrow V, W \]

\[ A \leftarrow \neg(V \land W) \quad \Rightarrow \quad A \leftarrow \neg V \text{ and } A \leftarrow \neg W \]
Lloyd Topor transformation ctd

Implication

\[ A \leftarrow V \rightarrow W \quad :\Rightarrow \quad A \leftarrow W \ \text{and} \ A \leftarrow \neg V \]
\[ A \leftarrow \neg(V \rightarrow W) \quad :\Rightarrow \quad A \leftarrow V, \neg W \]

Negation

\[ A \leftarrow \neg\neg V \quad :\Rightarrow \quad A \leftarrow V \]

Universal quantification

\[ A \leftarrow \forall x_1 V \quad :\Rightarrow \quad A \leftarrow \neg\exists x_1 \neg V \]
\[ A \leftarrow \neg\forall x_1 V \quad :\Rightarrow \quad A \leftarrow \exists x_1 \neg V \]

The transformations so far are easy to justify as giving logically equivalent clauses.
Lloyd Topor ctd

Existential quantification

\[ A \leftarrow \exists x_1 V \quad \Rightarrow \quad A \leftarrow V \]
\[ A \leftarrow \neg \exists x_1 V \quad \Rightarrow \quad A \leftarrow \neg p(y_1, \ldots, y_n) \text{ and } p(y_1, \ldots, y_n) \leftarrow \exists x_1 V \]

Here \( p \) is a newly introduced predicate, and \( y_1, \ldots, y_n \) are the free variables appearing in \( \exists x_1 V \).

When there are several formulas in the body, apply the transformations as expected, eg:

\[ A \leftarrow B, V \lor W, C \quad \Rightarrow \quad A \leftarrow B, V, C \text{ and } A \leftarrow B, W, C \]
Example

given:

\[
\text{subset}(X, Y) :\neg \forall u \ (\text{in}(u, X) \rightarrow \text{in}(u, Y)).
\]

transformation gives two clauses

\[
\text{subset}(X, Y) :- \neg p(X, Y).
\]

\[
p(X, Y) :- \neg \text{in}(U, Y), \text{in}(U, X).
\]

This gives in sicstus:

| ?- subset( [3,5], [6,5,4,3,2] ). |
yes

| ?- subset( [3,5], [] ). |
no

| ?- subset( [3,5], [6,5,4] ). |
yes

% WRONG answer
The problem

Most Prolog systems place no restriction on when negation as failure is used. It is sound to use it if the arguments are ground when a goal is called, but otherwise it can be unsound, as in the example above.

Example

\[ p(a) :- \text{not } q(X). \]
\[ q(a). \]

Without any restriction on negation as failure, 'not p(a)' succeeds – because p(a) fails, since ungrounded q(X) succeeds. But 'not p(a)' does not follow from the completion of the program!

There are systems that do implement a sound form of negation as failure, however; they are needed to execute general programs in a way that respects their logical meaning.
Dealing soundly with negation as failure

What needs to be done to get an interpreter that deals soundly with negation as failure?

• Negated goals should only be tested if they are ground (no variables);

• A goal with variables may become ground when later goals succeed (if variables are shared between goals).

• Can freeze goals with variables, and only call when and if they become ground. This is a useful mechanism in general.
Coroutining

The ability to suspend goals until some property holds is useful when the programmer has a good idea of how to achieve some result computationally.

See sicstus on co-routining, and built-ins, eg

```
when(+Condition,:Goal)
   Blocks Goal until the Condition is true, where
   Condition is a goal with restricted syntax combining:
       nonvar(X), ground(X), ==(X,Y)
```

Alan Smaill  Logic Programming  Nov 8, 2010
Summary

- Closed World Assumption & Negation as Failure.
- Clark completion
- Lloyd-Topor transformation