Today

School of informatics

- More on Closed World Assumption & Negation as Failure.
- Clark completion
- Higher-order logic programming

Reminder: Negation by failure

School of informatics

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.

Logic and monotonic reasoning

School of informatics

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.

$$\textbf{If } T \models Q, \textbf{ then } T \cup X \models Q$$

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 $poor(fred)$ to get the new T':

$$\begin{array}{rcl} & & poor(jane) \\ poor(jane) & \rightarrow & happy(jane) \\ & & happy(fred) \\ & & poor(fred) \end{array}$$

Now $\neg poor(fred)$ is not in $X_{T'}$, and so we do not have $CWA[T'] \models \neg poor(fred)$ any more.

Negation by Failure and Least Herbrand Model

School of informatics

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

```
?- p(t,v).
```

Recall that `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 `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.

Dealing soundly with negation as failure

School of informatics

Nagation as failure is more widely used in Prolog, however. What needs to be done to get an interpreter that deals soundly with negation as failure in general?

- 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, and more flexible than reordering clause bodies in the program itself.

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)
```

Here ?=/2 is a (sicstus) builtin:

```
?=(+Term1,+Term2)

Succeeds if Term1 and Term2 are identical terms, or
if they are syntactically non-unifiable.
```

Clark completion

When negation by failure is used with no restriction, 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 $\mathcal{S}$, and computing an extended set of predicate calculus statements $Comp(\mathcal{S})$, and extending the Backchain inference system with negation by failure, to recover the desired connection again.

Completion

Suppose that theory has a single formula $\texttt{foo(a)}$
– this formula is equivalent to $\forall \texttt{x}\ (\texttt{x} = \texttt{a} \rightarrow \texttt{foo(x)})$.
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 $\forall \texttt{x}\ \texttt{foo(x)} \rightarrow \texttt{x} = \texttt{a}$.
We now describe a procedure to calculate the completion of a set of definite clauses.

Completion procedure

School of informatics

- Suppose start with a definite clause
  $\forall \mathtt{y}\ ((\mathtt{Q}_1(\mathtt{y}) \wedge \ldots \wedge \mathtt{Q}_\mathtt{n}(\mathtt{y})) \rightarrow \mathtt{p}(\mathtt{t}))$
  where $\mathtt{t}$ may be a tuple of terms; we assume that $\mathtt{y}$ is the only variable in the body of the clause, if not quantify over **all** the variables in the body.
- Put this in the equivalent form
  $\forall \mathtt{y}\ \forall \mathtt{x}\ ((\mathtt{x} = \mathtt{t} \wedge \mathtt{Q}_1(\mathtt{y}) \wedge \ldots \wedge \mathtt{Q}_\mathtt{n}(\mathtt{y})) \rightarrow \mathtt{p}(\mathtt{x}))$.
- Put this in the equivalent form
  $\forall \mathtt{x}\ (\exists \mathtt{y}\ (\mathtt{x} = \mathtt{t} \wedge \mathtt{Q}_1(\mathtt{y}) \wedge \ldots \wedge \mathtt{Q}_\mathtt{n}(\mathtt{y})) \rightarrow \mathtt{p}(\mathtt{x}))$.

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

$$\begin{aligned} \forall x\ .E_1 &\rightarrow p(x) \\ \forall x\ .E_2 &\rightarrow p(x) \\ &\vdots \\ \forall x\ .E_m &\rightarrow p(x) \end{aligned}$$

  which can be combined to give

$$\forall x\ .((E_1 \vee E_2 \vee \ldots \vee E_m) \rightarrow p(x))$$

Completion ctd

School of informatics

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

$$\forall \mathtt{x}\ (\mathtt{p}(\mathtt{x}) \leftrightarrow (\mathtt{E}_1 \vee \mathtt{E}_2 \vee \ldots \vee \mathtt{E}_\mathtt{m})).$$

Example

Take the clauses:

$\forall \mathtt{x}\ (\mathtt{scottish(x)} \rightarrow \mathtt{british(x)})$
$\mathtt{british(fred)}$

To take the completion, we get first:

$$\forall \mathtt{x}\ ((\mathtt{scottish(x)} \vee \mathtt{x} = \mathtt{fred}) \rightarrow \mathtt{british(x)}).$$

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

$$\forall \mathtt{x}\ (\mathtt{british(x)} \leftrightarrow (\mathtt{scottish(x)} \vee \mathtt{x} = \mathtt{fred}).$$

Note that from the completion, assuming $\mathtt{dai} \neq \mathtt{fred}$, we can deduce $\neg\mathtt{british(dai)}$, which is not a logical consequence of 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

$$\forall \mathtt{x}\ \forall \mathtt{y}\ \forall \mathtt{z}\ \neg \mathtt{foo}(\mathtt{x}, \mathtt{y}, \mathtt{z})$$

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

School of informatics

- $\mathcal{S}$ follows logically from $Comp(\mathcal{S})$
  (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 *positive* information: for atomic statements A,

$$\mathcal{S} \models A \text{ if and only if } Comp(\mathcal{S}) \models A$$

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

Recall:

School of informatics

For *pure Prolog* (Prolog without meta-logical predicates), we have a declarative reading of a program as a logical description of a problem domain.

This uses definite clauses; in the program variables are (implicitly) universally quantified ($\forall$), and in queries existentially quantified ($\exists$). We search for a derivation that the the query follows logically from the program.

Contrast:

School of informatics

Functional programming languages (Haskell, ML, LISP) are also declarative, in a different way: the program specifies a meaning for each function
These languages are higher order, in that the functions themselves are first-class objects of the language, and can be passed around as arguments.

This is not the case for Prolog (in the declarative reading), since it corresponds to first-order logic. Some features of Prolog as a programming language allow a bit more (eg `call/1`).

Is there something analogous we can do with a Logic Programming approach?

What Higher-Order Logic Programming is not

School of informatics

We have already seen that we can use Prolog as a meta-language for Prolog, and so manipulate Prolog programs in Prolog.
This is not using higher-order ideas: it is mixing together two separate first-order representations, one a representation of an object domain, and another a representation of the syntax of the first representation.
The logic here is called first-order because quantifiers are only used over individual variables – we can't quantify over functions, or predicates in this logic.

A richer logic

Suppose we allow quantifiers also over predicates: this takes us to second-order logic. We extend the syntax of first order logic by allowing variables for predicates as well as for individuals, and all $\forall, \exists$ quantifiers using these variables. The reading of these quantifiers is just what you would expect ...

Example:

$$\forall P.\ P(0) \wedge \forall x\ (P(x) \rightarrow P(suc(x))) \rightarrow \forall y\ P(y)$$

This lets us express standard induction on the natural numbers as a single statement about all properties P.

Note that if we tried to express this directly in definite clause logic, there are three problems:

- Prolog variables can't appear in the "predicate" position (since it's first order).
- One of the subgoals is an implication.
- We want local quantification of the x.

What would be good . . .

School of informatics

We'd like to be able to write something like:

```
P(Y) :- P(0), ∀x.(P(x) => P(suc(x))).
```

– read this as:

```
P(Y) :- ( P(0), ∀x.(P(x) => P(suc(x))) ).
```

and have a programming language that made sense of this.

λProlog

School of informatics

We outline a language that lets us do this sort of thing. It should let us:

- Search for derivations systematically;
- Provide witnessing answers for query variables.

In the first-order case we have seen the unification algorithm that is used in computing solution values for query variables within definite clause logic; this has to be extended to deal with other kinds of variables.

λ-terms

School of informatics

Haskell, LISP and ML make use of λ-terms:

```
Haskell:   \x -> x + 4
LISP:      (lambda (x) (+ x 4))
ML:        fn x => x + 4
```

and the evaluation of the applications of such terms is the main computational mechanism of the languages.

λProlog includes such terms also, with the syntax

```
x\ (x + 4)
```

and the treatment of such terms has to let equivalent terms be equal (and find solutions).

Examples

School of informatics

```
?- (x\ (x + 4)) = (y\ (y + 4)).
solved

?- (x\ (x + 4)) 3 = 3 + 4.
solved

?- F 3 = 3 + 4.

F = x\ 3 + 4 ;

F = x1\ x1 + 4 ;

no more solutions
```

Extending the language

School of informatics

definite clause logic is extended by adding:

- A type structure: syntax items have user declared types; there is a special type o of propositions; functions from type t_1 to type t_2 have type $t_1 \rightarrow t_2$. Predicates on objects of type t have type $t \rightarrow o$.
- Implication as a new connective: `G => H`.
- Universal quantification (in programs and queries).
- Existential quantification (just in queries).

We use a curried style of syntax (as typical in Haskell), rather than tuples (as in Prolog).

Quantifiers

Use the following to express quantification:
for $\forall x\ A$, use a lambda term to express the binding of the variable, and then a constant `pi` to quantify. Thus a goal

$$\forall x\ x = x$$

becomes

```
pi ( x\ (x = x) )
```

and $\forall P\ P(0) \rightarrow P(0)$ becomes

```
pi ( p\ ( (p 0) => (p 0) )).
```

Both of these succeed as queries.

Examples

School of informatics

```
?- pi x\ (x = x).

solved
?- pi x\ ( pi y\ ( x = y)).

no
?- pi x\ (x = (Y x)).

Y = x\ x ;

no more solutions
```

Inferring with implications

School of informatics

An implication goal `?- P => Q` is tackled by "adding `P` to the program", and trying to show `Q`. Standard Prolog clauses allow just one implication (in the other direction).

```
a :- b.
b :- c.
?- c => a
Solved
```

Note that the statement added – here `c` – is added only locally in the context of the implication query; so backtracking must keep track of where assumptions are added, so that they are not in scope if backtracking takes the context back before this query.

Inferring with implications ctd

School of informatics

More complex statements can get added too:

```
a :- b.
c.
?- (c => b) => a
Solved
```

Here `c => b` is added, so this is equivalent to querying `?- a.` with the program:

```
a :- b.
c.
b :- c.
?- a.
Solved
```

This gives most of the expected properties of implication, e.g.

```
?- a => (b => a).
solved

?- (a => (b => c)) => (a => b) => (a => c).
solved
```

However, this is not implication as characterised by the standard truth table. Consider:

```
?- ((a => b) => a) => a.
no
```

Summary

School of informatics

- Closed World Assumption & Negation as Failure.
- Clark completion
- Higher-order logic programming