Admin

School of informatics

- Exam in December is only for students visiting Edinburgh for 1 semester.
- Dates of exams are now known. These are provisional, check for confirmed dates later.
- There is an on-line programming exam; and a short (1 hour) theory exam.

Today

School of informatics

- Predicate Calculus: syntax, semantics
- Definite Clause logic
- Substitutions and Unification
- Inference System of definite clause logic

Recall: propositional definite clauses

School of informatics

We saw use of propositional definite clauses for reasoning:

- Notion of logical consequence via interpretations;
- Notion of an inference system for proofs;
- Notion of proof search.

We have similar notions for the language of pure Prolog.

Properties

School of informatics

Correctness of Prolog proof search: straightforward, see later

$F_1, \ldots, F_n \vdash_{Prolog} g_1 \wedge \cdots \wedge g_m$ implies $F_1, \ldots, F_n \vdash g_1 \wedge \cdots \wedge g_m$

Soundness of Inference System: not so hard, see later

$F_1, \ldots, F_n \vdash g_1 \wedge \cdots \wedge g_m$ implies $F_1, \ldots, F_n \models g_1 \wedge \cdots \wedge g_m$

Completeness of Inference System: propositional case argued last time

$F_1, \ldots, F_n \models g_1 \wedge \cdots \wedge g_m$ implies $F_1, \ldots, F_n \vdash g_1 \wedge \cdots \wedge g_m$

Incompleteness of Prolog proof search:

take easy example

$F_1, \ldots, F_n \vdash g_1 \wedge \cdots \wedge g_m$ does not imply

$F_1, \ldots, F_n \vdash_{Prolog} g_1 \wedge \cdots \wedge g_m$

Logic Programming aims to use Predicate Calculus as a representation language, where programs have a **declarative reading** as logical theories – ie as sets of formulas of the Predicate Calculus. The main computation follows from searching for a derivation of a query from the theory. The slogan is:

> *A program is a theory over a formal logic, and its computation is deduction in that theory.*

To make this idea work effectively, a subset of predicate logic is taken.

Grammar for first-order logic

School of informatics

. . . aka predicate calculus

Define *terms* by

$$
\begin{array}{rcl}
\textit{term} & ::= & \textit{constant} \\
 & & \mid \textit{var} \\
 & & \mid \textit{fn\_symbol}\ \mathbf{(}\ \textit{term\_list}\ \mathbf{)} \\
\textit{term\_list} & ::= & \textit{term} \\
 & & \mid \textit{term}\ \mathbf{,}\ \textit{term\_list}
\end{array}
$$

Formulas (= making a statement)

School of informatics

$$
\begin{array}{rcl}
\mathit{form} & ::= & \mathit{pred}\ \mathbf{(}\ \mathit{term\_list}\ \mathbf{)} \\
& & \mid\ \neg\ \mathit{form} \\
& & \mid\ \mathit{form}\ \vee\ \mathit{form} \\
& & \mid\ \mathit{form}\ \wedge\ \mathit{form} \\
& & \mid\ \mathit{form}\ \longrightarrow\ \mathit{form} \\
& & \mid\ \forall\ \mathit{var}\ \mathit{form} \\
& & \mid\ \exists\ \mathit{var}\ \mathit{form}
\end{array}
$$

Use precedence to disambiguate (or brackets).

Syntax and disambiguation in Prolog

The syntax of Prolog does not distinguish between *function symbols* and *predicates* – but it is important when looking at the declarative reading of the language to make this distinction.
Prolog allows functors (ie predicates and function symbols) to be declared is infix. It allows the use of brackets to impose a reading, and also has a system of precedence that allows brackets to be omitted while imposing a unique reading of a statement.
To check on the parsing in (sicstus, and maybe other) Prolog, use `display/1`:

```
| ?- display(a + b * c = 4).
=(+(a,*(b,c)),4)
yes
```

School of informatics

We say what it is for a formula to be *true* under an interpretation in a structure.
Write $\mathcal{S}$ for a structure together with an associated interpretation $\mathtt{I}$.
Given $\mathcal{S}$, and a formula $\mathbf{F}$, write $\mathcal{S} \models \mathbf{F}$ for "$\mathbf{F}$ is true in $\mathcal{S}$".

Modelling the world

School of informatics

We suppose that the world can be describe in terms of

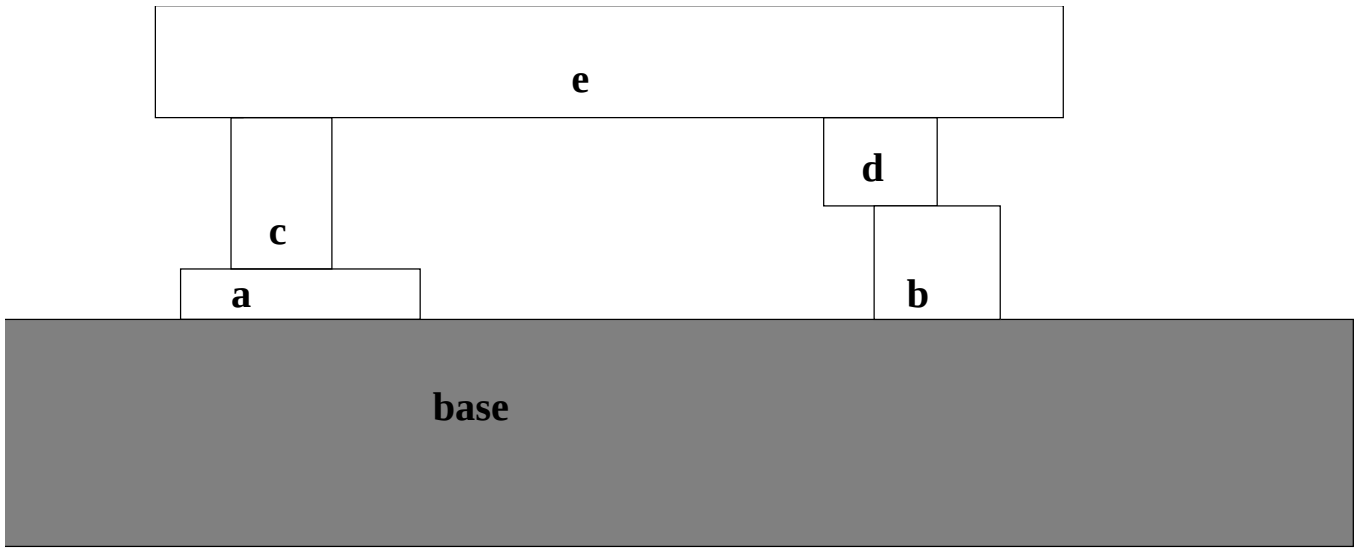
- set of objects
- functions
- relations

where

- functions map objects to objects
- relations with n arguments describe properties of n objects.

Example

Look at a blocks world:



This can be described using objects $\mathtt{O} = \{\mathtt{base}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$.
We can say which object is on which other one with a 2-place relation, *on*:
$\mathtt{on} = \{(\mathtt{e}, \mathtt{c}), (\mathtt{c}, \mathtt{a}), (\mathtt{e}, \mathtt{d}), (\mathtt{d}, \mathtt{b}), (\mathtt{b}, \mathtt{base}), (\mathtt{a}, \mathtt{base})\}$.

Structures

A structure is a set with some functions and relations.
We *interpret* a formula by giving an interpretation mapping:

- constants to objects
- function symbols to functions
- predicate symbols to relations.

Example

For the blocks world, fix the language:

- Constants: **base, a, b, c, d, e**.
- Function symbol: **next**
- Predicate: **on**

An interpretation for this is:

$$\begin{array}{lcllcl} I(\textbf{ base }) & = & base & I(\textbf{ a }) & = & a \\ I(\textbf{ b }) & = & b & I(\textbf{ on }) & = & on \\ & & \vdots & & & \end{array}$$

We *could* take $I(\textbf{ base }) = a$ however.

Satisfaction

We now say what it is for a formula to be true under an interpretation in a structure. Write $\mathcal{S}$ for a structure together with an associated interpretation $\mathtt{I}$.
Given $\mathcal{S}$, and a formula $\mathbf{F}$, write "$\mathbf{F}$ is true in $\mathcal{S}$" as:

$$\mathcal{S} \models \mathbf{F}$$

Suppose $\mathbf{P}$ is a predicate and $\mathbf{c}$ is a constant. To see if

$$\mathcal{S} \models \mathbf{P(c)}$$

look at $I(\ \mathbf{P}\ )$ and $I(\ \mathbf{c}\ )$:

$$\mathcal{S} \models \mathbf{P(c)} \qquad \text{iff} \qquad \mathcal{I}(\ \mathbf{P}\ )(\mathcal{I}(\ \mathbf{c}\ )).$$

Satisfaction ctd

Extend this to formulas with connectives:

$$
\begin{array}{lll}
\mathcal{S} \models \mathbf{A} \wedge \mathbf{B} & \textit{iff} & \mathcal{S} \models \mathbf{A} \;\; \textit{and} \;\; \mathcal{S} \models \mathbf{B} \\
\mathcal{S} \models \mathbf{A} \vee \mathbf{B} & \textit{iff} & \mathcal{S} \models \mathbf{A} \;\; \textit{or} \;\; \mathcal{S} \models \mathbf{B} \\
\mathcal{S} \models \mathbf{A} \rightarrow \mathbf{B} & \textit{iff} & (\; \textit{not}\, (\mathcal{S} \models \mathbf{A})) \;\; \textit{or} \;\; \mathcal{S} \models \mathbf{B} \\
\mathcal{S} \models \neg \mathbf{A} & \textit{iff} & \textit{not}\, (\mathcal{S} \models \mathbf{A})
\end{array}
$$

For example, with interpretation given,

$$
\begin{array}{lll}
\mathcal{B} & \models & \mathbf{on(c, a)} \wedge \mathbf{on(e, c)} \\
\mathcal{B} & \models & \neg(\mathbf{on(c, a)} \rightarrow \mathbf{on(e, a)}) \\
\textit{not}(\mathcal{B} & \models & \mathbf{on(a, c)} \vee \mathbf{on(c, c)})
\end{array}
$$

Roughly, the idea is that for any statement $\Phi(v)$ which talks about variable v:

$$\mathcal{S} \models \forall \mathbf{v_n}\ (\mathbf{\Phi(v_n)}) \quad \text{if and only if} \quad \mathcal{S} \models \mathbf{\Phi(v_n)}$$

for all interpretations of $\mathbf{v_n}$

$$\mathcal{S} \models \exists \mathbf{v_n}\ (\mathbf{\Phi(v_n)}) \quad \text{if and only if} \quad \mathcal{S} \models \mathbf{\Phi(v_n)}$$

for some interpretation of $\mathbf{v_n}$

Logical Consequence

School of informatics

Our semantics gives us a notion of *logical consequence* as before. We say that a formula $\mathbf{G}$ is a logical consequence of formulae $\mathbf{F_1}, \mathbf{F_2} \ldots \mathbf{F_n}$ (meaning that it follows logically) if and only if, for all structures with interpretation $\mathcal{S}$,

$$\textit{if } \mathcal{S} \models \mathbf{F_1} \textit{ and } \ldots \textit{and } \mathcal{S} \models \mathbf{F_n},$$
$$\textit{then } \mathcal{S} \models \mathbf{G}.$$

When this is true, we write

$$\mathbf{F_1}, \mathbf{F_2} \ldots \mathbf{F_n} \models \mathbf{G}.$$

Suppose we restrict the statements we consider as follows:

- Drop quantifiers (but keep variables)
- Drop $\neg$ $\vee$ (but keep $\wedge$, $\rightarrow$).
- Only allow formulas of the shape

$$p(t_1, \ldots, t_n)$$

for some predicate p (ie an atomic statement), or

$$A_1 \wedge \cdots \wedge A_n \rightarrow B$$

where each A_i is an atomic statement.

Formulas of these shapes are called **Definite Clauses**, and they have interesting properties for inference.
We can use our previous definition of logical consequence here; for statements involving variables, take the statement to be implicitly **universally** quantified.

The idea is that if knowledge can be expressed as a set of definite clauses S, and we are interested in knowing if some atomic statements follow, then we want to find out if

$$S \models A_1 \wedge \ldots A_n.$$

where variables in the query are implicitly **existentially** quantified.

Example

Take definite clauses as follows.

$$
\begin{array}{rcl}
 & & father(jon, kay) \\
 & & daughter(kay, liz) \\
father(X, Y) & \rightarrow & ancestor(X, Y) \\
daughter(X, Y) & \rightarrow & ancestor(Y, X) \\
ancestor(X, Y) \wedge ancestor(Y, Z) & \rightarrow & ancestor(X, Z)
\end{array}
$$

Then we can ask if it follows that $ancestor(jon, liz)$, or if there is a Q such that $ancestor(Q, kay)$: $\exists q.ancestor(q, kay)$.

Substitution

A *substitution* is formally as an association of terms for variables, *ie* a set of variable/term pairs. Thus $\{\ \mathtt{x/a}, \mathtt{y/g(w)}, \mathtt{z/b}\ \}$ is the substitution where the variable $\mathtt{x}$ is to be replaced by the term $\mathtt{a}$, the variable $\mathtt{y}$ is to be replaced by the term $\mathtt{g(w)}$, and so on.

A substitution is applied by replacing free occurrences of the variables by the corresponding terms. For a substitution $\mathtt{S}$ and a formula $\mathtt{F}$ we write $\mathtt{FS}$ for the result of applying the substitution. For the substitution above, we have for example

$$\mathtt{P(x, g(x), y)\{\ x/a, y/g(w), z/b\ \} = P(a, g(a), g(w))}$$

Substitution ctd

Substitutions are important, since the result of a successful logic program query **is** a substitution that associates a term with each of the (Prolog) variables that appear in the initial query. We will see how this substitution is characterised and computed.

Note that the substitutions for variables in S must be applied *simultaneously*, rather than one at a time.
Consider $\{ x/y, y/g(a) \}$ applied to $f(x)$; the result is $f(y)$, and *not* $f(g(a))$ which we would get by applying first the substitution $\{ x/y \}$ and then $\{ y/g(a) \}$.

Substitution ctd

We will want to combine substitutions into a single substitution to compute the result of a query. Write $S_1 \circ S_2$ for the substitution that applies S_1 first, and then S_2. The defining property of this composition operation is that it is always true that

$$t(S_1 \circ S_2) = (tS_1)S_2$$

The *identity* substitution $\{\ \}$ leaves terms untouched ($t\{\ \} = t$). So $S \circ \{\ \} = S = \{\ \} \circ S$.
Substitutions are just function applications, so we also always have

$$\begin{array}{rrcl} & t(S_1 \circ (S_2 \circ S_3)) & = & t((S_1 \circ S_2) \circ S_3) \\ \text{ie} & S_1 \circ (S_2 \circ S_3)) & = & ((S_1 \circ S_2) \circ S_3). \end{array}$$

School of informatics

We can now define what it is for two expressions to be *unifiable*. This occurs for formulae $\mathtt{F}, \mathtt{G}$ when there is a substitution θ such that $\mathtt{F}\theta = \mathtt{G}\theta$. We can think of this as saying that there are values for the variables that make the equation true (by making the terms identical). In this case, θ is said to be the *unifier* of the two expressions. In logic programming, unifiers are the basic objects to be computed during program execution.

We can define an order relation between substitutions: when is one substitution *more general* than another? Define $S_1 \preccurlyeq S_2$ to mean that there is a third substitution S such that $S_1 = S_2 \circ S$. We say that S_2 is *more general* than S_1.
Thus $\{\ x/f(a), y/g(a, f(b)), w/f(b)\ \} \preccurlyeq \{\ x/f(z), y/g(z, w)\ \}$ – what is the S here?

Most general unifiers

It turns out that in the case of the term syntax we have given, if two terms t_1, t_2 are unifiable, then there is a most general unifier (mgu) S. This means that

1. $t_1 S = t_2 S$
2. for any S', if $t_1 S' = t_2 S'$, then $S' \preccurlyeq S$.

The most general unifier is not unique as a substitution; however, any two mgus are equivalent in the following sense

If S_1, S_2 are mgus for t_1, t_2, then $S_1 \preccurlyeq S_2$ and $S_2 \preccurlyeq S_1$.

This last property in fact just means that $t_1 S_1, t_1 S_2$ just differ by renaming of variables back and forth.

School of informatics

The given definite clauses are taken as axioms. We use a single inference rule, Backchain:

$$\frac{p_1\theta,\ \ p_2\theta,\ \ldots,\ p_n\theta,\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{q'\theta}$$ where θ is mgu of q', q

Given a query $(\exists X) r(x)$, see if $r(X)$ unifies with the "head" formula of a definite clause, with unifier θ. If so, top-down search will look for justifications of $p_1\theta,\ \ p_2\theta,\ \ldots,\ p_n\theta$.

Example

School of informatics

For goal

$$ancestor(Q, kay),$$

we can use clause

$$daughter(X, Y) \rightarrow ancestor(Y, X)$$

with unifier

$$\{\ Y/Q, X/kay\ \}$$

to get subgoal

$$daughter(kay, Q).$$

Choice Points

School of informatics

If we use only this rule, then what are the choice points in the search?

- Which clause to pick;
  there may be several clauses whose heads match a given query. This is an **OR** choice: it is enough for any one such choice to succeed.
- The order to tackle the subgoals;
  this is an **AND** choice: all the subgoals must be shown, but the order may affect if a derivation is found, depending on the search strategy used.

The standard form of logic programming (Prolog and friends) is based on this fragment of first order logic.
The part of the language that corresponds to this reading of programs as logical statements and computation as inference in that logic is called pure Prolog — other features are needed to get a practical programming language out of this.
A Prolog clause

```
ancestor(X,Y) :- father(X,Z), ancestor(Z,Y)
```

is simply an alternative notation for

$$father(X, Z) \wedge ancestor(Z, Y) \rightarrow ancestor(X, Y)$$

Inference strategy

School of informatics

The strategy adopted in Prolog for search is to search depth-first, in a top to bottom (for the **OR** choices) and left to right for the **AND** choices.

- Top to bottom: pick the clauses in the order in which they appear in the program.
- Left to right: tackle the subgoals in the order in which they appear in the chosen clause.

Good News!

The backchain rule gives a **complete** inference system with respect to the standard semantics of first-order logic: for given clauses C and query Q, if $C \models Q\theta$ for some θ, then there is a derivation of $Q\theta$ using the backchain rule.

Bad News!

However: the Prolog inference strategy is **incomplete**: it may fail to find a derivation, even when a derivation exists. (Examples are easy to find.)

Bad News!

Also, it is known that there is **no** decision procedure for the problem of showing if a query follows or not from a set of definite clauses. (This is hard to show)

Summary

School of informatics

- Predicate Calculus: syntax, semantics
- Definite Clause logic
- Substitutions and Unification
- Inference System of definite clause logic