Today:

- Higher order logic for KR

Recall:

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 Horn 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

:

Functional programming languages (LISP, ML) 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.

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

What Higher-Order Logic Programming is not

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: is *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 (compare the reflection hypothesis).

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 Horn 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.

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

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

and have a programming language that made sense of this.

λProlog

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 there is a *unification* algorithm that is used in computing solution values for query variables (see Automated Reasoning module for details); this has to be extended to deal with other kinds of variables.

λ-terms

Both LISP and ML make use of λ-terms:

```
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

```
?- (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

Horn 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 $t_1 \to t_2$. Predicates on objects of type t have type $t \to o$.
- Add implications to the language: `G => H`.
- Universal quantification (in programs and queries).
- Existential quantification (just in queries).

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) )
```

(the outer brackets can be omitted); and $\forall P\ P(0) \to P(0)$ becomes

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


Examples

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

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

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

Y = x\ x ;

no more solutions
```

Search

What search operations are used to solve queries?

There are search operations associated with different connectives in the goal; for example:

- To solve `D => G`, add `D` to the program clauses, and solve `G`.
- To solve `pi (x\ G x)`, pick a new parameter `c` (i.e. a constant that does not appear in the current problem), and solve `G c`.
- To solve atomic `G`, find a program clause whose head can be instantiated to match `G`, and solve the body.

Example (McCarthy)

Try to formalise the following:

> Something is sterile if all the bugs in it are dead.
> If a bug is in an object which is heated, then the bug is dead.
> This jar is heated.
> *So,* the jar is sterile.

This is a natural and simple argument, and we want to express in directly. We could use full predicate calculus (but search is hard there).

In the language above, we get as follows.


```
type sterile (i -> o).
type in      (i -> i -> o).
type heated  (i -> o).
type bug     (i -> o).
type dead    (i -> o).
type j        i.

sterile Jar :- pi x\ ( (bug x) =>
                 (in x Jar) => (dead x) ).
dead X      :- heated Y, in X Y, bug X.
heated j.

% query:
% ?- sterile j.
% solved
```


Using higher-order features.

Often we want to do similar things for different predicates we are reasoning about. For example, the standard `ancestor/2` predicate is defined as a transitive extension of `parent/2`:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Similarly, get *less than* from the *successor* relation, *descendent* from *child* . . .

Now, do this once and for all:

```
type  trans (A -> A -> o) -> (A -> A -> o).

trans Pred X Y :- Pred X Y.
trans Pred X Z :- Pred X Y, trans Pred Y Z.
```

and define `ancestor` via

```
ancestor X Y :- trans parent X Y.
```


Inferring with implications

Recall that a 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
```

More complex statements can get added too:

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


Agent Inference

We can use this to give declarative accounts of different agent inference processes, in a common framework with inference about an object domain:

```
type base_bel     agent -> o -> o.     % primitive beliefs
type bel          agent -> o -> o.     % derived beliefs
type a1           agent.
```

Give all agents a simple inference mechanism:

```
bel A B :- base_bel A B.
bel A Q :- base_bel A (P => Q), bel A P.
bel A (P & Q) :- bel A P, bel A Q.
bel A (bel A X) :- bel A X.          % introspection
```

Suppose some facts about family relationships. We can then express:

```
% a1 has real facts, plus one extra belief.

base_bel a1 (parent X Y) :- parent X Y.
base_bel a1 (parent sean barney).

% a2 just has real facts

base_bel a2 (parent X Y) :- parent X Y.

% agents have standard notion of ancestor

base_bel A ( parent X Y => ancestor X Y ).
base_bel A ( ( parent X Y & ancestor Y Z ) => (ancestor X Z) ).
```


Examples

Now can query for a1's beliefs, which include the consequences of the "false" belief, unlike a2's beliefs:

```
?- bel a1 (ancestor sean X).

X = barney ;

X = liz ;

no more solutions
?- bel a2 (ancestor sean X).

no
```


More complex inference

Search becomes expensive quickly here. We can restrict the amount of inference the agents perform:

```
bel A (parent A B) :- parent A B.
                    % believe prolog!
bel A (bel A F) :- bel_base A F.
                    % limited introspection
bel A F :- inferrable F,
            % just look at "interesting" statements
           bel_base A G, bel_base A H,
           G => H => F.
            % limited deductive power
            %  (got from 2 basic beliefs).
```


Now we can express more complex relationships between belief systems.

```
parent a b.

% belief base
bel_base a (parent b c).
bel_base a ((parent X Y) => (ancestor X Y)).
bel_base a (pi x\ (bel b x) => (bel a x)).
       % a believes he believes
       % everything b believes.
bel_base b (parent c b).
       % b has this different from a.
```

Example queries

From this we get that agent a has some strange beliefs –

```
?- bel a (bel a (parent b X)).

X = c
?- bel a (bel a (parent c X)).

X = b
?- bel a (parent c X).

no
```


Summary

- Can extend Logic Programming paradigm to a richer language.
- Allows predicates to be arguments to (HO) predicates.
- Incorporates λ-term reduction.
- Has an associated notion of uniform search.