Logic Programming: Higher Order LP

Alan Smaill

Nov 15, 2012
Higher order logic programming
  - Extended syntax
  - Extended unification
  - Adapt notion of Uniform Search
Extending the logic

- Allow variables not just for individuals, but for predicates and function symbols also.
- Explicit quantifiers
- Allow “anonymous function patterns”

Languages like ML and Haskell have anonymous functions: haskell:

\( \lambda x \rightarrow x + 4 \)

ML:

\( \text{fn } x \rightarrow x + 4 \)
Mix with Prolog variables:

\[ x \backslash (x + Y) \]

Now unification has to take account of the binding of variables; there is no difference if bound variables are renamed; and a pattern applied to an argument is considered to match with the result of substituting the argument. Also such patterns can be the solutions to queries with variables corresponding to functions.
Examples

[toplevel] ?- (y\ y + 2) 3 = 3 + 2.

yes

[toplevel] ?- (X\ X + 4) = (y\ y + Z).

The answer substitution:
Z = 4

NB bound variables can be upper or lower case.
The language $\lambda$Prolog invented by Dale Miller is based on these ideas; see web site:

http://www.lix.polytechnique.fr/~dale/lProlog/

The unification algorithm is more complicated, but has the same aim, of looking for substitutions that when applied make formulae identical – taking into account bound variable renaming, and plugging in values in anonymous functions. The syntax is a bit different – uses curried style, familiar from Haskell. Because unification uses typing information, some type declarations are needed.
Example program signature

First, give signature (typing of predicates); the type \( o \) is the type of statements, so a predicate takes some number of arguments, and the return type is \( o \).

The file `reverse.sig`:

```
sig reverse.

type reverse (list A) -> (list A) -> o.
```

`reverse` is a predicate of two arguments, each of which is a list. The type of the elements of the list can be anything (but the same in both cases) – \( A \) acts like a Prolog variable.
Example program

reverse.mod:

module reverse.

type reverse (list A) -> (list A) -> o.
type reverse_aux (list A) -> (list A) -> (list A) -> o.

reverse L1 L2 :- reverse_aux L1 nil L2.
reverse_aux nil L2 L2.
reverse_aux (X::L1) Acc L2 :- reverse_aux L1 (X::Acc) L2.

Compile, link, and run as normal:

[reverse] ?- reverse [3,4,2,5] L.
The answer substitution:
L = 5 :: 2 :: 4 :: 3 :: nil
So far, this is a fussy way to write normal Prolog. We can exploit the higher-order features to write a map predicate; this takes a list and a function, and returns the result of applying the function to each member of the list. Because we have relations available, we can also think of mapping predicates (what could this mean?).

```
sig maps.

type mapfun list A -> (A -> B) -> list B -> o.
type mappred list A -> (A -> B -> o) -> list B -> o.
type for_each list A -> (A -> o) -> o.
```
Because we are in a relational, rather than functional, setting, what is available with the typing $A \rightarrow B$ is limited. Here’s the definition of $\text{mapfun}$:

\[
\text{mapfun nil F nil.}
\]
\[
\text{mapfun } [X|L] \text{ F } [(F X)|K] \text{ :- mapfun } L \text{ F } K.
\]

Notice the use of $F$ as a variable for a function – this goes beyond Prolog, and keeps reversibility.
Running mapfun

[maps] ?- mapfun [3,4,5] (x\ x + x) Y.

The answer substitution:
Y = 3 + 3 :: 4 + 4 :: 5 + 5 :: nil

?- mapfun X (x\ x + x) [3 + 3, 8 + 8].

The answer substitution:
X = 3 :: 8 :: nil
Here’s a definition for `mappred`; again note the variable standing for a predicate:

```
mappred nil P nil.
```

What will happen on back-tracking?

Suppose we have some background predicate:

```
father jane moses.   father john peter.
father jane john.    father james peter.
```
running mappred

[maps] ?- mappred [jane,john] father L.

The answer substitution:
L = moses :: peter :: nil

More solutions (y/n)? y

The answer substitution:
L = john :: peter :: nil

More solutions (y/n)? y

no (more) solutions
[maps] ?- mappred X father [moses,peter].

The answer substitution:
X = jane :: john :: nil

More solutions (y/n)? y

The answer substitution:
X = jane :: james :: nil

More solutions (y/n)? y

no (more) solutions
In practice (this implementation), it is not possible to query for a predicate where the query results in the predicate being called as a variable.

It is possible to use the higher-order features to indicate candidate predicates, though, and then search will find all possible predicates.
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

\[
\pi ( p \ ( (p \ 0) => (p \ 0) )).
\]

Here \( => \) indicates implication; this is a new form of goal, Prolog only allows conjunction.
Both of the above queries succeed.
Suppose we want to do some sort of reasoning about agents’ beliefs, where the agents may have some false beliefs, and where agents may introspect about their own beliefs. This is an example of how λProlog gives a good way to deal simultaneously with all these issues. Let’s do this in terms of basic and derived beliefs for a given agent. We introduce the property of a statement being inferrable – a second-order property. The signature introduces a new type of agent.
signature

sig bel.

kind agent type.

type bel agent -> o -> o. % derivable beliefs
type bel_base agent -> o -> o. % basis for belief set
type parent agent -> agent -> o.
type ancestor agent -> agent -> o.
type inferrable o -> o.

% some agents

type a agent. type b agent. type c agent.

end
Agent inference

Here is a simple way to allow inference for the agents (& is the built-in λProlog conjunction):

\[
\begin{align*}
\text{bel } A \; B & \; : \; \text{base}_\text{bel} \; A \; B. \\
\text{bel } A \; Q & \; : \; \text{base}_\text{bel} \; A \; (P \Rightarrow Q), \; \text{bel } A \; P. \\
\text{bel } A \; (P \land Q) & \; : \; \text{bel } A \; P, \; \text{bel } A \; Q. \\
\text{bel } A \; (\text{bel } A \; X) & \; : \; \text{bel } A \; X. \quad \% \text{ introspection}
\end{align*}
\]
Given a KB about family relationships, we can then express:

% a1 has real facts, plus one extra belief.

\[
\text{base\_bel a1 (parent X Y) :- parent X Y.}
\]
\[
\text{base\_bel a1 (parent sean barney).}
\]

% a2 just has real facts

\[
\text{base\_bel a2 (parent X Y) :- parent X Y.}
\]

% agents have standard notion of ancestor

\[
\text{base\_bel A ( parent X Y => ancestor X Y ).}
\]
\[
\text{base\_bel A ( ( parent X Y & ancestor Y Z ) => (ancestor X Z) ).}
\]
Querying agent’s beliefs

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
Different inference capabilities

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

bel A (parent A B) :- parent A B.
    % believe lambda 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).
Dependencies between agents’ 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
Next time

Next week, we will look at meta-programming in (standard) Prolog, and programming different forms of search than the built-in depth first search.