Logic Programming: Higher Order LP

Alan Smaill

Nov 14, 2013

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:

```
\x -> x + 4
```

ML:

```
fn x -> x + 4
```

HO patterns, extended unification

Mix with Prolog variables:

```
x\ (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.

λProlog

The language λProlog 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.
An implementation:

```
http://www.lix.polytechnique.fr/Labo/Dale.Miller/
                    lProlog/terzo/
```

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`.

```
% signature for 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.
```

This works as expected:

```
Terzo> #query lists.
?- reverse (3::2::1::5::nil) L.

L = 5 :: 1 :: 2 :: 3 :: nil
```

Beyond Prolog

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?).

```
type    mapfun    (A -> B) -> list A ->  list B -> o.
type    mappred   (A -> B -> o) -> list A -> list B -> o.
type    forevery  (A -> o) -> list A -> o.
```

mapfun

Because we are in a relational, rather than functional, setting, what is available with the typing `A -> B` is limited.
Here's the definition of `mapfun`:

```
mapfun F nil nil.
mapfun F (X::L) ((F X)::K) :- mapfun L F K.
```

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

Running mapfun

```
?-  mapfun (x\ x + x) (3::4::5::nil) Y.
```

```
Y = 3 + 3 :: 4 + 4 :: 5 + 5 :: nil
```

We can query for the **input** given the result:

```
?-  mapfun (x\ x + x) X (3 + 3 :: 8 + 8 :: nil).
```

```
X = 3 :: 8 :: nil
```

and even for the function, given inputs and outputs!

```
mapfun F (3::4::5::nil) (3+3 :: 4+4 :: 5+5 :: nil).
```

```
F = x\ x + x
```

mappred

Here's a definition for `mappred`;
again note the variable standing for a predicate:

```
mappred P nil nil.
mappred P (X :: L) (Y :: K) :- P X Y, mappred P L K.
```

What will happen on back-tracking?

Suppose we have a background predicate:

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

running mappred

```
?-  mappred father (jane :: john :: nil)  L.

L = moses :: peter :: nil ;

L = john :: peter :: nil ;

no more solutions
```

mappred reversible

```
?- mappred father X (moses :: peter:: nil).

X = jane :: john :: nil ;

X = jane :: james :: nil ;

no more solutions
```

Finding the predicate?

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.

Difference lists

Recall standard Prolog difference lists, which give an efficient way to do some list operations — and also need care in use.

In a higher-order setting, we can achieve the same efficiency gain, but remain declarative, and indeed retain reversibility.

The idea is that a normal list:

```
[1,3,5]
```

is represented by a **function** that maps any list to the list with `[1,3,5]` prepended; in Haskell syntax:

```
\x -> (1 : 3 : 5 : x)
```

Difference lists ctd

We can define functions to convert between the normal representation and this "difference" list version, and get an efficient way to append lists. Here are the type declarations; `list T` is a polymorphically typed list, and the difference lists have type `list T -> list T`:

```
type mkDList  list T -> (list T -> list T) -> o.

type append_dl
     (list T -> list T) ->
       (list T -> list T) ->
         (list T -> list T) -> o.
```

Difference lists ctd

These are implemented as follows:

```
% mkDList/2 uses standard recursion

mkDList nil (x\ x).
mkDList (H::T) (x\ H::(T' x)) :- mkDList T T'.
```

This works in both directions:

```
?- mkDList (1::3::5::nil) L.

L = x\ 1 :: 3 :: 5 :: x

?- mkDList L (x\ 1::3::5::x).

L = 1 :: 3 :: 5 :: nil
```

Difference list append

Now think a bit what corresponds to appending lists in this representation:

```
append_dl L M (x\ L (M x)).
```

So append is done via unification; we get reversibility here (there can be several unifiers, unlike in the usual Prolog situation).

Reversible append

```
?- mkDList (1::3::5::nil) L, append_dl L L Y.

L = x\ 1 :: 3 :: 5 :: x
Y = x1\ 1 :: 3 :: 5 :: 1 :: 3 :: 5 :: x1

?- mkDList (1::3::5::nil) L, append_dl X Y L.

L = x\ 1 :: 3 :: 5 :: x
X = x1\ 1 :: 3 :: 5 :: x1
Y = x2\ x2 ;

L = x3\ 1 :: 3 :: 5 :: x3
X = x4\ 1 :: 3 :: x4
Y = x5\ 5 :: x5 ;

% and another two solutions
```

Larger example

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`.

Type declarations

```
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.
```

Agent inference

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

```
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
```

Example ctd

Given a KB 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) ).
```

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 that 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
```