Logic Programming

Theory Lecture 8: Extensions of Predicate Logic

Richard Mayr

School of Informatics

10th November 2014

Overview

The theory half of the course has considered Prolog from the *declarative* perspective.

Under the declarative reading:

- Programs are logical theories
- The role of the system is to ascertain if the query is a logical consequence of the program.

This perspective accounts well for:

1. Definite clause logic, for both propositional logic (Lectures 1–2) and predicate logic (Lectures 3–5)

With some tuning of the basic ideas (completing the logical theory using CWA or Clark Completion), we also accounted for:

2. Negation by failure for negated ground queries (Lecture 7).

Moreover, items 1 and 2 above, can also be explained in terms of:

3. The role of the system is to establish *truth* in the minimal Herbrand model (Lecture 6).

Omissions

However, there are many aspects of Prolog we have not incorporated

- I/O (Programming Lecture 4)
- Cut (Programming Lecture 4)
- General negation as failure (Programming Lecture 5)
- `assert`, `retract`, ... (Programming Lecture 5)
- Term-manipulation primitives: `var`, `nonvar`, ... (Programming Lecture 9)

These omissions are for a fundamental reason.

The mentioned primitives do not fit in with the declarative reading.

For example, consider `assert`, in the context of an empty program.

```
?- p, assert(p)
```

These omissions are for a fundamental reason.

The mentioned primitives do not fit in with the declarative reading.

For example, consider `assert`, in the context of an empty program.

```
?- p, assert(p)
no
```

These omissions are for a fundamental reason.

The mentioned primitives do not fit in with the declarative reading.

For example, consider `assert`, in the context of an empty program.

```
?- p, assert(p)
no
?- assert(p), p.
```

These omissions are for a fundamental reason.

The mentioned primitives do not fit in with the declarative reading.

For example, consider `assert`, in the context of an empty program.

```
?- p, assert(p)
no
?- assert(p), p.
yes
```

These omissions are for a fundamental reason.

The mentioned primitives do not fit in with the declarative reading.

For example, consider `assert`, in the context of an empty program.

```
?- p, assert(p)
no
?- assert(p), p.
yes
?- p, assert(p)
```

These omissions are for a fundamental reason.

The mentioned primitives do not fit in with the declarative reading.

For example, consider `assert`, in the context of an empty program.

```
?- p, assert(p)
no
?- assert(p), p.
yes
?- p, assert(p)
yes
```

These omissions are for a fundamental reason.

The mentioned primitives do not fit in with the declarative reading.

For example, consider `assert`, in the context of an empty program.

```
?- p, assert(p)
no
?- assert(p), p.
yes
?- p, assert(p)
yes
```

- The order of atomic formulas in the query matters — contradicting the commutativity of $\wedge$ which is the declarative reading of comma.

These omissions are for a fundamental reason.

The mentioned primitives do not fit in with the declarative reading.

For example, consider `assert`, in the context of an empty program.

```
?- p, assert(p)
no
?- assert(p), p.
yes
?- p, assert(p)
yes
```

- The order of atomic formulas in the query matters — contradicting the commutativity of $\wedge$ which is the declarative reading of comma.
- The same query receives a different response depending on the state of the system.

Declarative extensions of Prolog

Today we consider two directions for extending Prolog that *do* fit in with the declarative reading.

- Extension from first-order logic to higher-order logic, as exemplified by λProlog

- Meta-programming. (non-examinable)

Declarative extensions of Prolog

Today we consider two directions for extending Prolog that *do* fit in with the declarative reading.

- Extension from first-order logic to higher-order logic, as exemplified by λProlog

  We consider this in some detail.

- Meta-programming. (non-examinable)

Declarative extensions of Prolog

Today we consider two directions for extending Prolog that *do* fit in with the declarative reading.

- Extension from first-order logic to higher-order logic, as exemplified by λProlog

  We consider this in some detail.

- Meta-programming. (non-examinable)

  We consider this briefly.

A richer logic

In *first-order* logic, quantifiers are only used over variables standing for individuals (elements of the universe) — we can't quantify over functions, or predicates in this 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

$$\forall \mathtt{P}.\, \mathtt{P}(0) \wedge (\forall \mathtt{X}.\, \mathtt{P}(\mathtt{X}) \rightarrow \mathtt{P}(\mathtt{s}(\mathtt{X}))) \rightarrow \forall \mathtt{Y}.\, \mathtt{P}(\mathtt{Y})$$

This lets us express standard induction on the natural numbers as a single statement about all properties $\mathtt{P}$

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.
- The quantification over `X` is local.

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

```
P(Y) :- P(0), ∀X.(P(X) => P(s(X))).
```

Which is to be read as

```
P(Y) :- ( P(0), ∀X.(P(X) => P(s(X))) ).
```

And we'd like to have a programming language that made sense of this.

λProlog

The λProlog language lets us do this sort of thing.

It lets 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 is extended to deal with other kinds of variables.

λ-terms

The functional languages 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).
(In λProlog, variables such as `x` are allowed to start with lower case letters)

Extending the logic

Definite clause logic is extended by adding:

- A type structure: syntax items have user declared types; there is a special type `o` of *propositions*, and another type `i` of individuals.
  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`.
  In particular, first-order unary predicates have type `i` $\to$ `o`.
- Add implications to the language: `G => H`
- Universal quantification (in programs and queries).
- Existential quantification (just in queries).

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

That is, we write `(p x y)` instead of `p(x,y)`.

Quantifiers

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

$$\forall \mathtt{X}.\, \mathtt{X} = \mathtt{X}$$

becomes

```
pi ( X\ (X = X) )
```

and $\forall \mathtt{P}.\, P(0) \rightarrow P(0)$ becomes

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

Both of these succeed as queries.

Examples

```
?- 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

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 goes before this query.

Inferring with implications ctd

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
```

Inferring with implications ctd

The implication symbol `=>` *associates to the right*. This means that the formula `A => B => C` is equivalent to `A => (B => C)`.

So the λProlog query:

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

can be written with fewer brackets as:

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

The search for this query first adds `a => b => c` to the program, which translates to the Prolog clause `c :- a,b`, and then adds `b` then `a`. So we end up with the query.

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

This procedure 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
```

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`. (We say that `c` is a *fresh* parameter.)
- To solve atomic `G`, find a program clause whose head can be unified with `G`, and solve the body.

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`. (We say that `c` is a *fresh* parameter.)
- To solve atomic `G`, find a program clause whose head can be unified with `G`, and solve the body.
  This step is just as in ordinary Prolog

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 it directly. We could use full predicate calculus (but search is hard there).

In λProlog, we can do this as follows.

Type declarations

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

Program

```
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, we define *less than* from the *successor* relation, and *descendent* from *child*, all using the same pattern of definition.

In λProlog we can 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.
```

We can use this to define `ancestor` via

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

Here the predicate `parent` is used as an argument to the `trans` procedure.

Meta-programming (non-examinable)

One further omission from the course so far has been a discussion of *meta-programming*

In contrast to the other omissions, meta-programming can (partially) be reconciled with the declarative reading.

("Partially" because some Prolog meta-primitives, such as `var/1` cannot be understood declaratively.)

We consider some aspects of meta-programming from a declarative viewpoint.

Meta-language

When we have two languages, and use one to talk about the other, we are using the first language as a *meta language* to talk about the second language, which is called the *object language*.

Examples

English as meta language, with French as object language

> *The word "poisson" is a masculine noun*

English as both meta and object language

> *It is hard to pronounce "Peter Piper Picked a Piece of Pickled Pepper"*

Meta language in Prolog

Prolog contains a mixture of object-level and meta-level statements.

So far, in the theory course, we have seen Prolog used as an object language. E.g.

```
father(a,b).
```

uses the `father` predicate to state that `a` is the father of `b`.

However

```
functor(father(a,b),father,2)
```

uses Prolog as a meta language to talk about itself. Here, we state that the Prolog term `father(a,b)` has `father` as its main operation and this has 2 arguments.

Meta-programming in Prolog

The ability of Prolog to talk about itself, for example describe its own syntax, has a number of uses.

For example, it is used crucially in *meta-programming* applications, such as writing an interpreter for Prolog in Prolog itself, or variations on similar themes, some very useful.

Prolog as a meta-language declaratively

The *declarative ideal* of interpreting programs as theories and queries as logical consequences does extend to cover those aspects of meta-programming that are independent of the current state of the system.

This is natural. For example, the statement

```
functor(father(a,b),father,2)
```

makes a perfectly declarative assertion.

However, there is a problem in using *definite clause predicate logic* as the basis of this declarative understanding. Logically, `father(a,b)` is a formula rather than a term, so cannot appear inside a surrounding predicate (here `functor`).

(This is not an issue in Prolog since it does not make a syntactic distinction between formulas and terms.)

Two possible solutions

1. Within definite clause predicate logic. Encode Prolog syntax (terms and formulas) as data, for example using lists (so `father(a,b)` would be encoded as `[father,a,b]`). Then meta-predicates can be expressed as ordinary predicates describing properties of the encodings.E.g.,

   ```
   myfunctor([father,a,b],father,2)
   ```

2. By separating the object and meta logic. This is the approach taken in the logic programming language Goedel:

   ```
   http://www.scs.leeds.ac.uk/hill/GOEDEL/expgoedel.html
   ```

But perhaps Prolog's choice of blurring the distinction between predicate and function symbols is the more practical one for programming.

Summary of course

Definite clause predicate logic provides the basis for understanding the declarative core of Prolog.

Other aspects of logic programming (restricted forms of negation by failure, higher-order logic programming, meta-programming) admit a declarative reading in extensions of definite-clause predicate logic.

Still other aspects of Prolog (I/O, cut, etc.) are essentially *procedural* and so do not fit into the declarative framework.

Summary of course

Definite clause predicate logic provides the basis for understanding the declarative core of Prolog.

Other aspects of logic programming (restricted forms of negation by failure, higher-order logic programming, meta-programming) admit a declarative reading in extensions of definite-clause predicate logic.

Still other aspects of Prolog (I/O, cut, etc.) are essentially *procedural* and so do not fit into the declarative framework.

Punchline (cf. Conclusions slide of Lecture 2)

Prolog (also λProlog) is an example of good engineering design based on an interplay between theoretical and practical considerations.

Main points today

Non-declarative features of Prolog

Higher order logic programming: λProlog

The extended logic of λProlog

Extending search for λProlog queries

Meta-programming (non-examinable)