Introduction to Prolog

Mark Armstrong

September 23, 2020

Contents

1 Introduction

These notes were created for, and in some parts **during**, the lecture on September 18th and the following tutorials.

2 Motivation

Today, we begin investigating another non-imperative paradigm other than functional programming: *logical programming*.

As we've seen in the previous hands-on lecture, functional programming takes advantage of *immutability* in order to make reasoning about programs easier. It also focuses on *compositionality*, including in its use of *higher-order functions*, all of which makes programming very much like writing functions in mathematics (where there is no mutable state).

Logical programming also assumes immutability, but instead of compositionality as a method of computation, it uses a (Turing-complete) subset of first-order predicate logic. Programs are databases of *inference rules* describing the problem domain, and programs are initiated by *queries* about the problem domain which the system tries to prove are true (a logical consequence of the rules) or false (refutable by the rules).

To put it simply, in logical programming, you describe the problem, rather than the solution.

3 A note about fonts

In these notes, I use plaintext blocks in order to write the inference rules, and use em-dashes (—) to create the horizontal rules.

In some cases, the em-dashes may not show quite correctly. For instance, in the PDF version of these notes, there is a small space between each dash. In some browsers, they may not show at all (they work in my install of Chrome, at least.)

Apologies for any issues reading these notes caused by this.

4 (Re)introduction to inference rules

Recall: an inference rule has the form

$$\frac{\texttt{Premise}_1 \quad \texttt{Premise}_2 \quad \ldots \quad \texttt{Premise}_n}{\hookrightarrow \quad \begin{array}{l}\texttt{Rule Name}\\ \texttt{Conclusion}\end{array}}$$

where $\texttt{Premise}_1$, $\texttt{Premise}_2$, ..., and $\texttt{Premise}_n$ are some statements in our domain, and `Conclusion` is a statement that can be concluded from the premise statements.

In the domains of logics, the statements range over formulae (i.e., boolean expressions built up from boolean constants, predicates and propositional connectors), and we may have rules such as

$$\frac{\texttt{P} \qquad \texttt{Q}}{\texttt{P} \wedge \texttt{Q}}\ \wedge\texttt{-Introduction}$$

which says "given `P` and `Q`, we may conclude `P` $\wedge$ `Q`",

$$\frac{\texttt{P} \wedge \texttt{Q}}{\texttt{P}}\ \wedge\texttt{-Elimination}^{l}$$

which says "given `P` $\wedge$ `Q`, we may conclude `P`", and

$$\frac{\texttt{P} \qquad \texttt{P} \Rightarrow \texttt{Q}}{\texttt{Q}}\ \texttt{Modus Ponens}$$

which says, by translating the $\Rightarrow$ to English, "given `P` and if `Q` follows from `P`, then we can conclude `Q`".

(Technically, these are *rule schemas*; the *meta-variables* `P` and `Q` can be instantiated to obtain specific rules.)

Note that in these rules, we have the following *meta-syntax*:

1. Whitespace between premises is understood as a form of conjunction.
2. The horizontal rule is understood as a form of implication.

Any rule which does not have a premise is called an *axiom*; axioms are the known results of our domain, which do not need to be proven. For instance,

$$\frac{}{\texttt{true}}\ \top\texttt{-Introduction}$$

A collection of inference rules (or rule schemas) and axioms gives us a *proof system*.

See, for instance, the natural deduction proof calculus for classical logic.

You have likely seen the *equational* approach to proofs which is favoured by Gries and Schneider's "A Logical Approach to Discrete Math", and used in CS/SE 2dm3 at McMaster using the CalcCheck tool. Proof systems are an alternate approach to proof; see Musa's notes on the relationship from this year's 2dm3.

Inside of a proof system, we may construct proofs of statements via *proof trees*, which are trees where every node is a statement, and the connections between nodes correspond to the use of rules.

For instance, we have the following silly proof of `true` which uses the rules given above.

$$\dfrac{\dfrac{}{\texttt{true}}\ \top\texttt{-Introduction} \qquad \dfrac{}{\texttt{true}}\ \top\texttt{-Introduction}}{\hookrightarrow\ \dfrac{\dfrac{\texttt{true} \wedge \texttt{true}}{\texttt{true}}\ \wedge\texttt{-Elimination}^{l}}{\texttt{true} \wedge \texttt{true}}\ \wedge\texttt{-Introduction}^{l}}$$

Notice, by convention, we write proof trees from the **bottom up**. The root, at the bottom, is what we intend to prove. The leaves, at the top, must either

1. be axioms, or
2. be local assumptions.

Proof trees may be read from the top down, to see how the conclusion follows from the axioms and assumptions. It is generally better to read from the bottom up, though; otherwise the proof often seems to make unwarranted steps, or informally, it "pulls a rabbit from its hat".

5 Inference rules in other domains

The use of inference rules is not limited to the domain of logics.

It is perhaps better not to think of inference rules as defining a *proof system* (which makes us think of truth values and logics), but as defining a *game*: starting from the rules and axioms, what can we obtain?

For instance, consider the following problem.

5.1 The two bucket problem

5.1.1 The problem

Suppose you are given two buckets,

- one of which holds 5 units of water, and
- one of which holds 3 units of water.

You are tasked with collecting exactly 4 units of water; no more, and no less. You begin with 0 units in both buckets.

You may at any point

- fill one bucket entirely from a tap,
- pour the water out of a bucket, emptying it entirely, or
- pour one bucket into another until either the first is empty or the second is full.

You are tasked with collecting exactly 4 units of water using only those three kinds of actions.

5.1.2 The rules

Let us represent the state of the bucket by a pair of numbers. In these rules, we will use

- `x` as a meta-variable for the amount in the bucket which can hold 5 units, and
- `y` as a meta-variable for the amount in the bucket which can hold 3 units.

We can begin only if we have 0 units in both buckets.

$$\frac{}{\texttt{0 , 0}}\ \texttt{Start}$$

The action of filling a bucket replaces its current amount with the maximum amount.

$$\frac{\texttt{X , Y}}{\texttt{5 , Y}}\ \texttt{Fill}^{l} \qquad \frac{\texttt{X , Y}}{\texttt{X , 3}}\ \texttt{Fill}^{r}$$

The action of emptying a bucket replaces its current amount with 0.

$$\frac{\texttt{X , Y}}{\texttt{0 , Y}}\ \texttt{Empty}^l \qquad \frac{\texttt{X , Y}}{\texttt{X , 0}}\ \texttt{Empty}^r$$

:TODO:

$$\frac{\texttt{X , Y}}{\texttt{X + D , Y - D}}\ \texttt{Pour}^l\ \texttt{(provided X + D} \leq \texttt{5} \wedge \texttt{Y - D} \geq \texttt{0} \wedge \texttt{(X + D = 5} \vee \texttt{Y - D = 0))}$$

$$\frac{\texttt{X , Y}}{\texttt{X - D , Y + D}}\ \texttt{Pour}^r\ \texttt{(provided X - D} \geq \texttt{5} \wedge \texttt{Y + D} \leq \texttt{0} \wedge \texttt{(X - D = 0} \vee \texttt{Y + D = 3))}$$

6 Prolog

6.1 Programs are databases of inference rules

Prolog programs are simply databases of inference rules, also called *clauses*. An inference rule

$$\frac{\texttt{A}_1 \quad \texttt{A}_2 \quad \ldots \quad \texttt{A}_n}{\texttt{B}}$$

is written in Prolog as the clause

```
b :- a1, a2, ..., an.
```

(the ... is pseudocode, not Prolog syntax) (notice the period ending the rule). As with our inference rule, this rule states that `b` is true if all of the $\texttt{a}_i$ are true. So we can think of `:-` as $\Leftarrow$, and `,` as $\wedge$.

An axiom

$$\frac{}{\texttt{C}}$$

can be written in Prolog as

```
c :- true.
```

or, more simply,

```
c.
```

6.2 Get "output" by making queries

To interact with Prolog programs, we make *queries*, to which Prolog responds by checking its inference rule database to determine possible answers.

For instance, list catenation in SWI Prolog is a ternary predicate

```
append(X,Y,Z)
```

the rules of which enforce that `Z` is the result of catenating `X` and `Y`.

So we can make queries about catenation, such as

```
% Is [1,2,3,4,5,6] the result of catenating [1,2,3] and [4,5,6]?
?- append([1,2,3], [4,5,6], [1,2,3,4,5,6])
% Response:
% true.


% What are all the possible values of Z for which
% Z is the catenation of [1,2,3] and [4,5,6]?
?- append([1,2,3], [4,5,6], Z)
% Response:
% Z = [1, 2, 3, 4, 5, 6]


% What are the possible values of X and Y so that,
% when they are catenated, the result is [1,2,3,4,5,6]?
?- append(X, Y, [1,2,3,4,5,6])
% Response:
% X = [],
% Y = [1,2,3,4,5,6] ;
% X = [1],
% Y = [2,3,4,5,6] ;
% ...
```

Note: in this way, we get several "functions" from one predicate, depending upon what question(s) we ask!

6.3 Names, kinds of terms

In Prolog, any name beginning with an upper case letter denotes a variable.

Names which begin with lower case letters are *atoms*, which are a type of constant value. Atoms may be used as the name of predicates.

Character strings surrounded by single quotes, `'`, are also atoms. So we can write

```
'is an empty list'([]).
```

As you would expect, Prolog also has numerical constants, such as `1` or `3.14`.

Aside from variables and constants, the remaining kind of Prolog term is a *structure*, which has the form

```
atom(term1, ..., term1)
```

(again, the ... is pseudocode.)

6.4 Interacting with Prolog

As we've said, a Prolog program consists of clauses (inference rules.) For instance

```
head(X) :- body(X,Y)
```

which can be interpreted as having meaning

$\forall$ `X` $\cdot$ `head(X)` $\Leftarrow$ ($\exists$ `Y` $\bullet$ `body(X,Y)`)

Then during computation, given this clause and the goal `head(X)`, the Prolog runtime is tasked with finding a substitution for `Y` which makes `body(X,Y)` true.

We provide Prolog with goals through queries, usually by loading our programs into the interactive query REPL, either by running

```
$ swipl my-program.pl
```

from the command line, or

```
?- consult('my-program.pl')
```

once running SWI Prolog. We can also `assert` or `retract` rules in the query REPL, if needed. (Also, use `listing` or `listing(name)` to see all given clauses or clauses about the `name` predicate.)

6.5 Unification

The computation model of Prolog involves *unification* of terms. Terms unify if either

1. they are equal, or
2. they contain variables that can be instantiated in a way that makes the terms equal.

So in general, unification involves *searching* for possible variable bindings, by making use of the clauses and *modus ponens* $((P \wedge P \Rightarrow Q) \Rightarrow Q)$.

6.5.1 The goal list

Through this process, the single goal presented by a query will usually turn into a collection of goals; for instance, if we query `?- p(5).` and the search uses a clause

```
p(X) :- q(X), r(X).
```

then we now have as goals `q(5)` and `r(5)`.

6.5.2 Backtracking

Consider the program

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

b.
c :- false.
```

As the Prolog runtime tries to prove `a`, it will use the first clause, and `fail` (because in trying to prove `c`, it reaches `false`, which it cannot prove.) At that point, it has to *backtrack*, and try a different clause to prove `a`.

In general, the runtime will backtrack several times during a proof, and it keeps track of which clauses have been tried.

:TODO: Aside: contradictory clauses

```
a :- false.
a.
```

6.5.3 SWI Prolog's search strategy

1. Attempt to apply clauses in order from top to bottom (as in the source code.)
    - Only backtrack and try other clauses after success or failure.
2. Perform a depth first search trying to prove each goal.
    - So if the current goals are `b` and `c`, try to prove `b` before considering `c`.

6.5.4 Examining the search strategy

In order to interactively see the process Prolog is using during unification, use the `trace.` command in the REPL. Then each query will result in a log of calls made and failures encountered.

6.6 Equality

Prolog does have an equality comparison , written simply `=` (not `==`). **However**, this equality does no simplification. So, for instance, if a variable `X` has been unified to value `5`,

```
X = 5
```

would be `true`, but

```
X = 2 + 3
```

would be `false`.

This non-simplifying equality allows us to consider the *construction* of terms, rather than just their value. But in case we want to actually carry out arithmetic or calculate other values, we can use the `is` comparison.

```
X is 2 + 3
```

will evaluate to `true`.

6.7 Exerting control over the search; the cut operator

In part because Prolog's searching mechanism can be naive, the programmer is given a certain amount of control over the search.

The most important mechanism for controlling the search that we will see is the *cut*.

A cut is written `!`, and can be understood as "no backtracking is allowed to go beyond this point".

6.8 Failure

In order to force a search to fail, we can use a strategic cut along with the `false` predicate (which cannot be proven), as in

```
p :- !, false.
```

When this clause is used, the cut is encountered, preventing any backtracking. Then, we immediately state the goal `false`, which cannot be proven. So, the runtime must give up here, and return `false`.

Note that we are not really specifying the return value by writing `false`; instead, we are exerting our control over the search to ensure a `false` result.

6.9 Trying to write the two bucket problem

In an ideal world, we would be able to almost directly translate the above inference rules into Prolog, like so.

```
buckets(0,0).
buckets(5,Y) :- buckets(_,Y).
buckets(X,3) :- buckets(X,_).
buckets(0,Y) :- buckets(_,Y).
buckets(X,0) :- buckets(X,_).
buckets(DX,DY) :-
  DX is X + D, DY is Y - D,
  buckets(X,Y),
  DX <= 5,
  DY >= 0,
  DX is 5 or DY is 0.
buckets(DX,DY) :-
  DX is X - D, DY is Y + D,
  buckets(X,Y),
  DX >= 0,
  DY <= 3,
  DX is 0 or DY is 3.
```

(Note the `_` is used where we would have a "singleton" variable, i.e., a variable which appears nowhere else in a clause. The `_` is simply an anonymous variable name, and using it reassures Prolog that we didn't mean to refer to another variable.)

But using our knowledge of SWI Prolog's search strategy, we can quickly see a problem with the order of these clauses. :TODO:

6.10 Checking for divisors of a number

```
hasDivisorLessThanOrEqualTo(_,1) :- !, false.
hasDivisorLessThanOrEqualTo(X,Y) :- 0 is X mod Y, !.
hasDivisorLessThanOrEqualTo(X,Y) :- Z is Y - 1,
↪  hasDivisorLessThanOrEqualTo(X,Z).
```

7 Resources

- Prolog cheat sheet, by Musa Al-hassy.
  - Includes links to several other resources.
- Lecture notes on Prolog by Joseph Goguen, UC San Diego.