Notes for Lecture 23 (Fall 2022 week 11, part 1): Modes in Prolog

Jana Dunfield

November 21, 2022

The code for this lecture is in `lec23.pl`.
These notes introduce one new topic, *modes*, and one specific feature of Prolog, printing.

1 Modes

By itself, Prolog code doesn't do anything. For something to happen, we have to run queries.

In Haskell, like most languages, a function is a machine that takes input (one or more arguments) and returns output. Functions cannot be run backwards; if you want the inverse of the function

```
add1 :: Integer -> Integer
add1 x = x + 1
```

you need to write another function:

```
subtract1 x = x - 1
```

As we'll see later, not all arithmetic functions have inverses, and the inverse *map* is not always easy to define, so let's look at a somewhat easier situation: a function whose input and output are elements of a Haskell data type.

```
data Colour = Orange
            | Rose
            | Violet
            | White
            | Grey
            | Black
            deriving (Show, Eq)

cycle :: Colour -> Colour
cycle Orange = Rose
cycle Rose   = Violet
cycle Violet = Orange
cycle _      = Grey
```

We might try to translate this into Prolog as:

```
cycle(orange, rose).
cycle(rose,   violet).
cycle(violet, orange).
cycle(_,      grey).
```

This behaves the same as the Haskell function for *some* queries, but not all. If all our queries look like

```
?- cycle( some-colour, Result).
```

where `some-colour` is a specific colour like `orange` or `black`, *and* if we stop at the first solution (by typing a period), we get the same answers we would get in Haskell:

```
?- cycle(rose, Result).
Result = violet .           % I typed a .

?- cycle(grey, Result).
Result = grey.              % Prolog finished by itself
```

In the first query, `cycle(rose, Result)`, Prolog waited for us to tell it whether to look for more solutions: it had already noticed that the last clause `cycle(_, grey)` matched.

In the second query, `cycle(grey, Result)`, Prolog finished by itself because only one clause matched (the last one: `cycle(_, grey)`).

If we type a semicolon instead, we get two results, and the second is not consistent with the Haskell code (`cycle Rose` returns `Violet`, not `Grey`):

```
?- cycle(rose, Result).
Result = violet             % I typed a ;
Result = grey.
```

We have two approaches to solving this problem.

The first approach is to "split" the wildcard into its possible Haskell values. Since the wildcard clause in the Haskell version of `cycle` comes after the Orange, Rose, and Violet patterns, the Haskell clause will match White, Grey, and Black. That leads to the Prolog code

```
cycle(orange, rose).
cycle(rose,   violet).
cycle(violet, orange).
cycle(white,  grey).
cycle(grey,   grey).
cycle(black,  grey).
```

Now, Prolog will give only one answer—the same one Haskell would have given—for queries like `cycle(rose, Result)`.

The second approach is to use cuts, which we will not discuss until Week 12.

For queries like `?- cycle(X, grey).` Prolog will give three solutions, corresponding to the three possible inputs to the Haskell function that return `Grey`. Technically, the Haskell function `cycle` does not have an inverse *function* because there is no unique `x` such that `cycle x == Grey`, but it has an inverse *map* that "returns" any of `White`, `Grey`, `Black`.

If we give the Prolog query

```
?- cycle(X, grey).
```

and ask for multiple solutions, we get the three solutions

```
X = white  % type ;
X = grey   % type ;
X = black.
```

1.1 Input and output modes

Queries involving the predicate `cycle` work regardless of which arguments to `cycle` are "concrete" (Prolog terms like `orange` and `grey`), and which arguments are "unknown" (Prolog variables like `X`):

```
?- cycle(orange, rose).   % Both arguments concrete; in Haskell:
                          %                           cycle Orange  ==  Rose
true.

?- cycle(orange, grey).   % Both arguments concrete; in Haskell:
                          %                           cycle Orange  ==  Grey
false.

?- cycle(orange, Z).      % Second argument unknown; in Haskell:
                          %                           cycle Orange
Z = rose.

?- cycle(X, orange).      % First argument unknown; Haskell can't do this
X = violet.

?- cycle(X, grey).        % First argument unknown; Haskell can't do this
X = white
X = grey
X = black.

?- cycle(X, Y).           % Both arguments unknown; Haskell can't do this
X = orange,
Y = rose
X = rose,
Y = violet
X = violet,
Y = orange
X = white,
Y = grey
X = Y, Y = grey
X = black,
Y = grey.

?-
```

When an argument is concrete, it is in *input mode*. When an argument is unknown (a variable), it is in *output mode*. Queries like

```
?- cycle(orange, Z).
Z = rose.
```

are the only kind that Haskell can handle directly, because the first argument `orange` is an input, and the second argument `Z` is an output. That is what the Haskell function does.

All argument "modings" (a moding is a sequence of modes, like "first argument input, second argument output") work for `cycle`, but that does not hold for all Prolog predicates.

For example, our factorial function only works when the first argument is input.

```
/*
  Factorial function

  ?- fact(5, R).
  R = 120
  false.
*/
fact(N, 1) :- N =:= 0.

fact(N, R) :- N > 0,
              NMinus1 is N - 1,
              fact(NMinus1, Result),
              R is N * Result.
/*
  ?- fact(X, 120).
  ERROR: Arguments are not sufficiently instantiated
  ERROR: In:
  ERROR:    [9] _2864>0
  ERROR:    [8] fact(_2890,120) at ...
  ERROR:    [7] <user>
*/
```

On the other hand, the built-in `append` predicate works for all modings, though it isn't particularly useful when all three arguments are outputs.

2 Printing

In Prolog, unlike Haskell, it is pretty easy to print messages while queries are being run.

The most versatile way to print is the built-in predicate `format`, which is basically similar to `printf` in C, and to the % operator in Python.

Here is an example from lec24.pl:

```
get2( Xs, (Y1, Y2)) :-
    print("a string~n"),
    print(Xs),
    member(Y1, Xs),
    format("called member, Y1 = ~p...~n", Y1),
    member(Y2, Xs),
    format("called member again, Y1 = ~p, Y2 = ~p...~n", [Y1, Y2]).
```

In general, `format` takes two arguments.

The first argument to `format` is a string, with `~p` denoting things we want to print (similar to `\%s` in some languages) and `~n` denoting a line break.

The second argument to `format` is a list of the things we want to print. (If there is only one thing to print, then we can give that thing, as above where we passed `Y1` as the second argument.)

(The `print` predicate also prints, but it can only print one thing, either a fixed string like `"a string~n"` or a Prolog thing like `Xs`.)

3 Arithmetic

It makes sense that the factorial predicate fails for "interesting" modes. A query like

```
?- fact(X, Y).
```

says "come up with some `X` and `Y` such that `fact(X) = Y`." Prolog does not have types, so we have to look at how `fact` is defined. But `fact` is defined using arithmetic operations that only work "forwards": if `X` is known, we can compute `Y`, but not backwards.

Some functions do not have inverses. For example, the function `halve` that divides by 2 (rounding down) does not have an inverse:

```
halve :: Integer -> Integer
halve x = x `div` 2

double :: Integer -> Integer
double x = x * 2
```

The function `double` is not an inverse. If we have a function f and its inverse f^{-1}, then for all x, we should have $f^{-1}(f(x)) = x$. The function `halve 21` returns `10`, but `double 10` returns `20`.

```
 -1
f     (f    (x))   =  x

double(halve(21))  =  20
```

If we want to "run `halve` backwards" on 10, we should really get both 20 and 21. That would require running arithmetic operations backwards, and in general, doing algebra to solve equations. This is more than Prolog can handle (which seems fair, it's not Matlab or Mathematica).

Thus, when we translate Haskell functions involving arithmetic, our translations will pretty much always work only in one mode.

4 On a quiz

When asking you to write a Prolog predicate from an English specification, I try to be clear about which modes the Prolog predicate needs to handle.

If the Prolog predicate needs to do arithmetic, there will usually just be one “moding”, like “first argument input, second argument input, third argument output”.

When translating a Haskell function to a Prolog predicate, you *never* need to handle modes that are “more unknown” than what the Haskell function can handle.