Notes for Lecture 24 (Fall 2022 week 11 part 2): Prolog "cut"

Jana Dunfield

November 23, 2022

The code for this lecture is in `lec24.pl`.

1 Controlling Prolog searching

(Some of this section is duplicated from previous notes. If it seems familiar, it's probably safe to skim it.)

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

In Haskell, pattern matching goes from the first clause defining a function to the last. As soon as the patterns match, Haskell starts running the right-hand side of the clause.

When we enter a query, Prolog also looks for matching clauses from top to bottom. Unlike Haskell, Prolog does not necessarily stop with the first clause. As we have seen, we often have to think about this when we translate Haskell functions to Prolog predicates. A common technique in Haskell is to write a functi
think about which argum

Here is an extended vers cture. Instead of only cycling between the three colours Orange/Rose/Violet, th
additional constructors in the type `Colour`: White, Grey, let cycle still exists, but when `cycle` is passed White, Grey, or Black, it retur

In Haskell, this can be implemented economically by adding a si `Grey`. Haskell reaches this clause only when the argument doesn't match any of Orange/Rose/Violet.

```
data Colour = Orange
            | Rose
            | Violet
            | White
            | Grey
            | Black

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

We might try to translate this into Prolog:

```
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 in
Haskell code (`cycle Re`


```
?- 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)`.

This approach has two clear advantages, and one disadvantage. Its advantages are:

- It is more clear than the second approach (cuts, discussed below): it requires writing more Prolog code, but the extra code is ordinary Prolog code that does not do anything strange.
- It is robust: it can handle queries with different modes.

The disadvantage is that it requires writing more code. Here, this approach added only two extra clauses, but for functions with many arguments, or complex arguments, this approach can add a *lot* of extra code.

(A variation on this approach is a single clause with a disjunction of equations:

```
cycle(orange, rose).
cycle(rose,   violet).
cycle(violet, orange).
cycle(X,      grey) :- X = white; X = grey; X = black.
```

Prolog interprets the semicolon ; as disjunction ("or"), so the last clause will be true if the argument `X` equals `white`, or equals `grey`, or equals `black`.)

The second approach is to put something in our code that will force Prolog to stop looking for another solution.

Before explaining the second approach, I would like to discuss my feelings about Prolog.

1.1 Prolog: both good and bad

Like any language, Prolog has parts I like and parts I don't. Some of the things I don't like (how arithmetic is done) could be fixed. Some other things I don't like are more fundamental to logic programming.

The attraction of logic p
rules (which I can easily tra
use to prove things withou zing.

Unfortunately (or fortunately?), from learning how to do pro
are only partly mechanical. Some of the steps we did in a 204 proof di t:
"we are trying to prove an implication, so let's use ing
where, when, and how to use the law of excluded middle (LEM) is som
no algorithm that always knows when and how to use LEM. We can program an algorithm that sometimes figures it out, but not always.

Prolog's search algorithm is not magic. It moves through the loaded file's clauses in order, looking for something that matches the query; if it does, and the clause is a rule, it tries to prove the body (premises) of the rule. If something doesn't work, it looks for another clause that matches and tries that.

The promise of logic programming is that I can write the rules and not think about how or when to use them. The inevitable disappointment of logic programming is that what Prolog does by default does not always work.

We can often "fix" this by adding information to the clauses (we are about to see how to do this) that tells Prolog when to stop looking for more solutions. This can be very useful. It will let us more easily translate Haskell functions, for example. But it betrays the promise of logic programming: I have to think about how Prolog will search through the rules.

If I have to think that much about Prolog's search algorithm, I'd probably prefer to use Haskell.

1.2 Cuts

To tell Prolog to stop looking for more solutions, we can use "cuts".

A cut is written "`!`". Roughly, it says to Prolog: "Forget about unexplored paths."

Here is what we get for `cycle`. I renamed it to `newcycle` so we can try both `cycle` and `newcycle` without having to comment out one set of clauses. I had to turn the first three facts into rules, because you can only use a cut in a rule. A cut is not really a premise of a rule in any logical sense, but it goes in the same place as a premise would.

Facts don't have premises, so we have to turn facts into rules to use cuts.

```
newcycle(orange, rose) :- !.
newcycle(rose,   violet) :- !.
newcycle(violet, orange) :- !.
newcycle(_,      grey).
```

Whereas `cycle(rose, Result)`, returned both `violet` and `grey`, the query `newcycle(rose, Result)` only returns `violet`:

```
?- newcycle(rose, Result).
Result = violet.
```

Why did this happen?

Prolog found the clause `newcycle(rose, violet) :- !.` and said, "To use this rule, I need to prove all its premises. The only premise I need to prove is `!`. That's a cut, so it's not really a premise. I have no premises left to prove. And I saw the cut, so I won't look for more solutions."

The Prolog code now f

1.3 Disappointment

I now have generalized sadness that I had to tell Prolog how to search. I

But I have a more specific disappointment.

A cool thing about Prolog is that when we implement Haskell fun often implementing the inverse relation *for free*. Look at `cycle` (*not* `newcycle`!): We can run queries that run `cycle` in reverse.

```
?- cycle(Input, violet).
Input = rose.
```

This isn't magic; it has some limitations. The response to this query isn't terribly satisfying:

```
?- cycle(Input, grey).
true.
```

I can't blame Prolog for this, however: the fact

```
cycle(_,      grey).
```

says that, given *any* input, the result is `grey`. So when I ask Prolog to find an `Input` such that `cycle(Input, grey)` is true, how should it know what I want?

We did this with no extra code. In Haskell, we could try to write a separate function:

```
inv_cycle Rose   = Orange
inv_cycle Violet = Rose
inv_cycle Orange = Violet
...
```

That wouldn't even work, though, because what do we return for `inv_cycle Grey`? `Grey` is returned by `cycle White`, `cycle Grey`, and `cycle Black`. We'd have to return a list of colours, instead of a single colour; then we could return `[White, Grey, Black]`. But this is already getting complicated.

Exercise 1. The original version of `cycle` from lec16, which does not include the colours White/Grey/Black, *does* have a straightforward inverse in Haskell. Why?

In Prolog, we just got it for free.

What happens with `newcycle`? Do we still get the inverse relation?

```
?- newcycle(Input, violet).
Input = rose.
?- newcycle(Input, grey).
true.
```


Seems to work as well a

With `cycle`, we can g ariables, and type semicolon repeatedly to get a lot


```
?- cycle(Input, Result).
Input = orange,
Result = rose

Input = rose,
Result = violet

Input = violet,
Result = orange

Result = grey.
```

I added blank lines between each solution—otherwise, the last one is extra confusing because (as explained above) Prolog has no idea what `Input` is there, and doesn't print a result for it at all.

Does *that* work with `newcycle`?

```
?- newcycle(Input, Result).
Input = orange,
Result = rose.
```

No. Prolog stopped immediately, because it saw the cut in the first clause.

1.4 What exactly does it mean to "forget unexplored paths"?

In `cycle/newcycle`, we added cuts to facts, so we had no choice about where to put the cut. In a rule, does it matter where we put the cut?

Yes. A cut prevents Prolog from backtracking: exploring other ways to try to make a query true. A cut only affects the past: any choices *already made* are "frozen". The cut does *not* directly affect choices to be made in the future.

See the discussion of `select_twice` in `lec24.pl`.

1.5 When should you use cuts?

A cut that affects the set of possible solutions to queries is called a *red cut*. A cut that does not affect the set of possible solutions to queries is called a *green cut*. However, even a green cut makes code harder to understand: we have to convince ourselves that the cut is green.

It is often convenient to use cuts when translating Haskell code: if we want to mimic the behaviour of the Haskell code, we never want more than one solution.

But even when translating Haskell code, we don't have to use a cut.

When writing code that is meant to produce many solutions (like `group` on Assignment 4), cuts should be used with extreme caution. We want `group` to backtrack whenever necessary, so it can find all the groups in the tree; any use of a cut is liable to stop it from finding all the groups.

1.6 More adventures wi

See `lec24.pl`.