Notes for Lecture 18 (Fall 2022 week 9, part 1): More on Prolog

Jana Dunfield

November 6, 2022

Code for this lecture is in `lec18.pl`.

1 Overview

What Prolog does when you run a query is similar to backwards-only proof search from CISC 204, but instead of the logical rules used in 204, Prolog uses the clauses (facts and rules) defined in the loaded Prolog file.

In this file, we explain how to think about Prolog clauses as logical formulas (Section 2), how to think about Prolog queries as logical formulas (Section 3), and how to translate Haskell code to Prolog (Section 5).

2 Translating clauses

2.1 Translating clauses to logical formulas

To translate a clause (a fact or rule) to a formula, we do the following:

- Find all the variables (words beginning with a capital letter) V1, V2, ..., and write $\forall$ V1, V2, ... (You can also write $\forall$V1 $\forall$V2 ..., with one $\forall$ for each variable.)
- If the clause is a fact (no `:-` symbol), just copy it (after the $\forall$ ...). (You should leave out the `.` ending it, since that's not part of a logical formula, but it's okay to leave it in.)
- If the clause is a rule (contains the `:-` symbol), remember that `:-` is a "backward" implication (the symbol is supposed to look like $\leftarrow$). We need to flip the parts of the rule around so we can write the usual implication $\rightarrow$.

  If there is more than one goal (after the `:-`, which means to the left of $\rightarrow$), we also need to turn the commas separating them into $\wedge$.

Some examples:

1 `delicious(logic).` becomes `delicious(logic)`

2 `knows(X, X).` becomes $\forall$X `knows(X, X)`

3 `knows(noether, X) :- strange(X).`
becomes $\forall$X (`strange(X)` $\rightarrow$ `knows(noether, X)`)

4 `knows(Y, Z) :- knows(Z, Y), knows(Y, noether).`
becomes $\forall$Y, Z (`knows(Z, Y)` $\wedge$ `knows(Y, noether)` $\rightarrow$ `knows(Y, Z)`)

Mistakes to avoid:

- Treating Prolog "atoms" like `logic` as if they were variables. The fact `delicious(logic)` says that logic is delicious.

  If I had written `delicious(Logic)`, the capital `L` would make `Logic` a variable, and *then* the translation would be $\forall$`Logic delicious(Logic)`—which says that everything is delicious.

- Not reversing `:-`.

- Not changing the commas between goals into conjunctions ($\wedge$). Something like `knows(Z, Y), knows(Y, noether)` is not a formula.

  (In 204, in the sequent $p, \neg q \vdash \neg\neg p$, there are two premises p and $\neg q$. The comma separates the premises; it is not part of the premise. Here, we want to get a single formula representing the rule, so we need to use $\wedge$.)

- Changing the commas *within* goals into conjunctions. For example, writing $\texttt{knows(Z} \wedge \texttt{Y)} \wedge \texttt{knows(Y} \wedge \texttt{noether)}$ instead of $\texttt{knows(Z, Y)} \wedge \texttt{knows(Y, noether)}$ is not correct. The formula `knows(Z, Y)` is a single proposition, like $K(z, y)$ might be in 204; it describes a relation between `Z` and `Y`. It does not join two formulas `Z` and `Y`.

One thing that would be considered wrong in 204, but which I will overlook, is not parenthesizing after the $\forall$s. According to the "operator precedence" in 204, the formula

$$\forall \texttt{X}\ \big(\texttt{strange(X)} \rightarrow \texttt{knows(noether, X)}\big)$$

must have the outer parentheses: in 204, if we wrote

$$\forall \texttt{X}\ \texttt{strange(X)} \rightarrow \texttt{knows(noether, X)}$$

the scope of $\forall$`X` would be `strange(X)`, and the second `X` would be free.

In 360, we will assume $\forall$ has "low precedence", and treat

$$\forall \texttt{X}\ \texttt{strange(X)} \rightarrow \texttt{knows(noether, X)}$$

as equivalent to

$$\forall \texttt{X}\ \big(\texttt{strange(X)} \rightarrow \texttt{knows(noether, X)}\big)$$

The order of the quantifiers doesn't matter (in 204, you may have seen the equivalence $\forall X\ \forall Y\ \varphi \equiv \forall Y\ \forall X\ \varphi$), but it seems most logical to write them in the same order they appear in the Prolog clause.

2.2 Translating clauses into mathematical English

It is possible to translate clauses into formulas without being told what the predicates are supposed to mean: `knows(noether, X)` becomes `knows(noether, X)`, regardless of what `knows` means.

If I want to translate a clause into what I call mathematical English, something like

> For all X, if X is strange then Noether knows X.

I need to be told that `knows(P, Q)` means "`P` knows `Q`". (It probably helps to be told that `strange(X)` means "`X` is strange", but that one is easier to guess.)

To translate a clause into mathematical English, I recommend first translating it to a formula. That tells you what the quantifiers are, and if the clause is a rule, it reorganizes the implication into the usual "if. . . then . . . ".

For example, "3" above:

```
knows(noether, X) :- strange(X).
```

becomes the formula

$$\forall \mathtt{X}\ \big(\mathtt{strange(X)} \rightarrow \mathtt{knows(noether,\ X)}\big)$$

To translate *that* to mathematical English, we translate the symbols $\forall$, $\wedge$, $\rightarrow$ into their English meanings ("for all", "and", "if . . . then"), and translate the predicates according to whatever we have been told.

If we did this halfway, we would might get:

For all X, if `strange(X)` then `knows(noether, X)`.

But this doesn't count as mathematical English, because it has some parts that are not English at all; what we want is

For all X, if X is strange then Noether knows X.

(If you wrote "noether" instead of "Noether", that would be okay, though I imagine she would prefer that her name be capitalized.)

3 Translating queries

Translating queries is almost the same as translating facts, with one very important exception: in a query, the variables (like `X`) are existentially quantified ("there exists"), *not* universally quantified.

So, while the fact

```
strange(X).
```

should be translated to $\forall \mathtt{X}\ \mathtt{strange(X)}$, the query

```
?- strange(X).
```

should be translated to $\exists \mathtt{X}\ \mathtt{strange(X)}$.

The fact says that everything is strange; the query is asking "is there something that is strange?".

When translating a query, don't try to run the query. If I ask you to translate

```
?- outrageous(Z).
```

you don't need to look for a fact or rule that concludes that something is `outrageous`. You only have to write $\exists \mathtt{Z}\ \mathtt{outrageous(Z)}$.

4 Writing Prolog

More specifically, I could give you some facts and rules in English, then ask you to write Prolog clauses that model those facts and rules.

5 Translating Haskell to Prolog

I like asking this kind of question better than just "writing Prolog", because instead of reading English description that might be ambiguous, you're given Haskell code that is unambiguous.

Some aspects of translating Haskell to Prolog work out nicely. For example, we can often turn each clause of a Haskell function into one Prolog clause. That means we don't have to understand the entire Haskell function at once; we can look at its clauses, one at a time, and turn them into Prolog clauses one at a time.

Other aspects can be more troublesome.

5.1 The result becomes an extra argument

Prolog predicates are either true or false. They can't return integers, trees, or other kinds of data. They are either true, or not. To represent a function that returns, say, a tree, we have to add an extra argument to the predicate.

For example, suppose we have a Haskell function that tells us whether a colour is "warm" or "cool".

```
data Colour = Orange
            | Rose
            | Violet
data Temperature = Warm
                 | Cool

temp :: Colour -> Temperature
temp Orange = Warm
temp Rose   = Warm
temp Violet = Cool
```


We can't make a Prolog predicate `temp` return `warm` or `cool` the same way as Haskell does; we have to add an extra argument that plays the role of the function result:

```
temp(C, T) iff Haskell temp C would return T
```

For the function `temp`, that's basically it.

```
temp(orange, warm).
temp(rose,   warm).
temp(violet, cool).
```

I was careful to turn the Haskell data constructors `Orange`, `Warm`, etc. into lowercase in Prolog. Anything that begins with an uppercase letter in Prolog is a variable, which means it's translated to a formula with a "for all". So if I wrote

```
temp(orange, Warm).
```

I would actually be saying that orange is both warm and cool, because the *variable* `Warm` could be replaced with anything, including `warm` and `cool`. If I wrote

```
temp(Orange, cool).
```

that would say that the temperature of everything is cool, because `Orange` would be a variable.

A Haskell function of two arguments becomes a Prolog predicate with three arguments, a Haskell function of three arguments becomes a Prolog predicate with four arguments, and so on.

(Haskell functions that return `Bool` *can* be handled without adding an extra argument, but we'll talk about that later.)

5.2 Function calls

Again, Prolog predicates can't return stuff, so it's not obvious how to translate a Haskell function that makes a function call.

```
data Colour = Orange
            | Rose
            | Violet
data Temperature = Warm
                 | Cool

cycle :: Colour -> Colour
cycle Orange = Rose
cycle Rose   = Violet
cycle Violet = Orange

cycle2 :: Colour -> Colour
cycle2 c = cycle (cycle c)
```


The function `cycle` returns the "next colour" of the three defined, and the function `cycle2` returns the "next next colour" by calling `cycle` twice.

We can translate `cycle` to Prolog in a similar way as we translated `temp`:

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

But if we try to translate `cycle2` in the same way, we would get

```
cycle2(C, cycle(cycle(C))).
```

This is (perhaps disturbingly) valid Prolog: it will think we're using `cycle` as a *data constructor*, and will conclude that the result of `cycle2` on `C` is the data structure whose syntax tree would be

```
cycle
  |
cycle
  |
  C
```

That's not what we meant—we're trying to call the function `cycle`.

We want to return the result of calling `cycle` twice on `C`. We need to phrase this as an "if-then" statement, because that's what Prolog lets us write. As an "if-then" statement, the Haskell clause

```
cycle2 c = cycle (cycle c)
```

says: "If the result of `cycle c` is `d`, and the result of `cycle d` is `e`, then the result of `cycle2 c` is `e`."

To see this more easily, we rewrite the Haskell clause to use "let", which gives a name to an expression:

```
cycle2 c =
  let d = cycle c in
  let e = cycle d in
    e
```

Now the structure of the Haskell function body has the same structure as "If the result of `cycle c` is `d`, and the result of `cycle d` is `e`, then the result of `cycle2 c` is `e`.".

Prolog makes us write implications right-to-left, so let's first rewrite the English sentence to "The result of `cycle2 c` is `e` if the result of `cycle c` is `d` and the result of `cycle d` is `e`."

We are modelling "the result of `cycle2 blah` is `something`" as `cycle2(blah, something)`, so we can write at least the first part of the Prolog clause by translating "The result of `cycle2 c` is `e`":

```
cycle2(C, E)
```


Next, we add the `:-` that represents "if":

```
cycle2(C, E) :-
```


Now we need to translate the stuff after "if".

"the result of `cycle c` is d" becomes `cycle(C, D)`.

"the result of `cycle d` is e" becomes `cycle(D, E)`.

The "and" becomes a comma.

This gives us

```
cycle2(C, E) :- cycle(C, D),
                cycle(D, E).
```

This behaves the same way as the Haskell code. For example, in the Haskell code, `cycle2 Orange` returns `Violet`, and we get `violet` from the Prolog query `cycle2(orange, X)`:

```
?- cycle2(orange, X).
X = violet.
```