Notes for Lecture 19 (Fall 2022 week 9 part 2): More on Prolog queries; arithmetic in Prolog

Jana Dunfield

November 6, 2022

Code for this lecture is in `lec19.pl`.

1 Backwards proof search

1.1 Analogy to 204

In CISC 204, you searched (manually) for proofs using both forward and backward reasoning. For example, to prove the sequent

$$p, q, q \rightarrow r \vdash p \wedge r$$

you could work backwards, writing $\wedge$i as the last step, then trying to prove p and r:

	p	
	r	
	$p \wedge r$	$\wedge$i

To prove p, we notice that we have p as a premise. The notice that $q \rightarrow r$ is a premise and prove q.

1	p	premise
	$q \rightarrow r$	premise
	q	
	r	$\rightarrow$e
	$p \wedge r$	$\wedge$i 1, __

1	p	premise
2	$q \rightarrow r$	premise
3	q	premise
4	r	$\rightarrow$e 2,3
5	$p \wedge r$	$\wedge$i 1, 4

This is backwards reasoning: we looked at our goal (what we want to prove), $p \wedge r$, and decided to use $\wedge$i as the last step. That gives us two new goals: p and r.

Alternatively, we could observe that q and $q \rightarrow r$ are premises, and prove r even before we decide to use $\wedge$i. That would be reasoning forward.

1	q	premise
2	$q \rightarrow r$	premise
3	r	$\rightarrow$e 1,2

...

Prolog is based on backward reasoning. (So is the “Tiny Theorem Prover” on assignment 3.) When you enter a query into Prolog, that is the goal. Prolog looks for a clause (fact or rule) that matches the goal. If the matching clause is a fact, the query is true. (Imagine trying to prove $q \vee \neg q$ in 204. As soon as we realize that we can use LEM, we are done. The LEM “rule” has no premises, so it is like a Prolog fact.)

If the matching clause is a rule, the situation is similar to using the $\wedge$e rule: the “body” of the Prolog rule—its premises, after the symbol `:-`—become our new goals.

In 204, reasoning (backwards or forwards) was done using a fixed set of rules: $\wedge$i, $\vee$e, $\rightarrow$i, and so on. In Prolog, the set of rules is not fixed: it depends on which `.pl` file you’ve loaded.

For example, if we are representing provability with a predicate `provable` and conjunction formulas using the constructor `conj`, we can model the $\wedge$i rule

$$\frac{\varphi_1 \qquad \varphi_2}{\varphi_1 \wedge \varphi_2}\,\wedge\text{i}$$

as a Prolog rule:

```
provable(conj(Phi1, Phi2)) :- provable(Phi1), provable(Phi2).
```

Exercise 1. Write one or two Prolog rules to model the $\vee$i1 and $\vee$i2 rules, as we modelled $\wedge$i above:

(Writing one rule requires using `;` within the head of the rule. It’s fine to write two rules.)

1.2 Prolog goes in order

Prolog always looks for matching clauses in order: it checks the first clause in the file first, then the second, and so on. So, as with Haskell’s pattern matching, the order in which we write clauses matters.

For example, the Haskell functions `f` and `g` have the same clauses, but they are written in a different order, so they behave differently: `f True` returns `True`, but `g True` returns `False`. The wildcard pattern in `g _ = False` matches `True`, so we return `False` and never reach the second clause `g True = True`.

```
f True = True
f _    = False

g _    = False
g True = True
```

(Haskell will warn you about `g` because the second clause is never reached, but it is just a warning, not an error.)

Translating the above Haskell functions to Prolog predicates leads to two predicates that behave differently, because Prolog also goes in order:

```
f(true, true).
f(_,    false).

g(_,    false).
g(true, true).

Results of queries (typing . to get only the first result):

?- f(true, Answer).
Answer = true .

?- g(true, Answer).
Answer = false .
```

But the story is a little more complicated for Prolog. As indicated above, I typed a period because I only wanted the first answer. If I had typed a semicolon, Prolog would have kept looking for more answers:

```
?- f(true, Answer).
Answer = true           % I typed a semicolon here
Answer = false.
```

```
?- g(true, Answer).
Answer = false          % I typed a semicolon here
Answer = true.
```

This isn't really what I wanted—Haskell functions always re ing to faithfully translate f and g to Prolog, they should only giv

(Mathematically, a binary relation $\mathcal{R}$ is a function if and only if, whenever (arg, answer1) and (arg, answer2) are both in $\mathcal{R}$, answer1 = answer2: given the argument arg, a mathematical function can't return answer1 sometimes, and answer2 other times. In this sense, Haskell functions are closer to mathematical functions than functions in many languages.)

There are several ways to fix this in the Prolog code and get Prolog predicates that really do correspond exactly to the Haskell code.

One way is to change the patterns in the Haskell code, before we translate, in a way that does not change the behaviour of the Haskell code, but makes the order of clauses irrelevant. Doing this for the Haskell function f, and calling the new function f2, we get:

```
f2 True  = True
f2 False = False     -- pattern was _, but we only reach this clause
                     --  when the argument does not match True,
                     --  so the pattern False is equivalent
```

When we translate f2, we get only one answer to our query:

```
f2(true,  true).
f2(false, false).
```

```
?- f2(true, Answer).
Answer = true.    -- Prolog noticed that there were no more matching
                  --  clauses, and didn't even wait for me to type . or ;
```

Exercise 2. Translate g to g2.

Hint: When Haskell is looking at the pattern in the first clause, which Bool values match that pattern?

The second way to do this is to leave the Haskell code alone, and do "the same thing" to the Prolog code: change the pattern _ to `true`. This would give us the same predicate `f2`.

There is a third way, called a *cut*, but we will not cover that right now.

2 Arithmetic

2.1 Dear Prolog, please do arithmetic

I hope the earlier notes have at least conveyed the idea that functions in Prolog don't look much like functions in other languages: we have to model functions as predicates.

Arithmetic operations like addition, subtraction, and multiplication are functions—we often write them differently (fo han prefix, like + 1 2), but they are essentially fun

If Prolog functions do d if Prolog arithmetic doesn't look li s like Haskell arithmetic, but getting Prolog to *do* arithmetic requires care.

To extract from Prolog the result of the function `f2` icate and then ask: "Does there exist `Answer` such that `f2, Answer` is true?"

So we might expect that, if we ask, "Does there exist `N` $N = 2 + 2$?", we would get 4. We would be wrong:

```
?- N = 2 + 2.
N = 2+2.

?-
```

What is going on here?

Prolog does not really have functions, so it is not going to just see `2 + 2` and convert it to 4. When it sees `2 + 2`, it thinks of that as *data*—as a tree:

```
    +
   / \
  2   2
```

This tree has one branch node, +, and two leaves, 2 and 2. If you saw this in a data structures course, it would not make sense to say this is really the tree

```
    4
```

because those are two different trees—they don't even have the same number of leaves.

Here is the `ArithExpr` data type and Haskell `calc` function from `lec8.hs`. It's been a while, so you'll probably want to go back to that file and remind yourself of how `calc` works:

```
-- Data type representing "arithmetic expressions",
-- which are also trees of a particular kind.
data ArithExpr = Constant Integer
               | Negate ArithExpr
               | Add ArithExpr ArithExpr
               | Subtract ArithExpr ArithExpr
               deriving (Show)

-- calc :: ArithExpr -> Integer
-- "Calculate" the result of an ArithExpr.
-- Examples:
--   calc (Negate (Constant (-5))) should step (eventually) to 5
--   calc (Negate (Negate (Constant (-5)))) should step (eventually) to -5
--   calc (Add (Constant 3) (Constant 4)) should step (eventually) to 7

calc :: ArithExpr -> Integer
calc (Constant k)      = k
calc (Negate e)        = 0 - (calc e)
calc (Add e1 e2)       = (calc e1) + (calc e2)
calc (Subtract e1 e2) = (cal
```

The function `calc` takes an `ArithExpr`, and returns a vant example, `calc (Add (Constant 2) (Constant 2))` returns 4.

This is not quite the same as how we wrote the tree in Prolog: I wo `(Add (Constant 2) (Constant 2)`
as

```
         Add
        /   \
       /     \
 Constant   Constant
    2          2
```

which doesn't look exactly like

```
      +
     / \
    2   2
```

But the differences are minor: in Prolog, we don't need to write `Constant`, and we write `+` instead of `Add`.

If writing `2 + 2` makes Prolog generate a tree, how do we tell Prolog to do the kind of thing that `calc` does?

For reasons unclear to me, `calc` in Prolog is spelled "is". If we write `is` instead of `=`, Prolog *will* do arithmetic:

```
?- N is 2 + 2.
N = 4.
```

Maybe Prolog uses "`is`" because people say things like "two plus two is four"? However, "`is`" only works in one direction: if we put the `2 + 2` on the left, and the `N` on the right, we get an error.

```
?- 2 + 2 is N.
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:    [8] 2+2 is _3378
ERROR:    [7] <user>
?-
```

Rant: I find it really annoying that Prolog makes you write the result `N` to the *left* of the input `2 + 2`. Translating functions to predicates with the argument first, and the result second, is consistent with how we usually represent functions as sets in mathematics. We haven't covered lists yet, but putting the result last is also consistent with how *Prolog* implements the 'append' function on lists; try `?- append([1, 2], [3], Answer).` So if Prolog actually had a function called `calc`, I'd expect to write `calc(2 + 2, N)`.

To summarize: to make Prolog do what `calc` does, use `is`, with a variable to the left of `is`.

2.2 More arithmetic in Pr

For some more arithmeti