Propositional logic and recursive datatypes

Patrick D. Elliott

May 22, 2023

- Haskell's type system can be used to represent grammars as datatypes.
- Grammars of non-trivial formal languages/natural languages typically involve *recursive* rules for well-formed formulas/sentences
- An example from a CFG for natural language:

```
S -> NP VP
VP -> V S
```

- "She is upset."
- "Zach said she is upset."
- "Jenna believes Zach said she is upset."

- In order to implement recursive grammars in Haskell, we use *recursive datatypes*.
    - Operations that apply to inhabitants of recursive datatypes are (typically) recursive functions.
    - Today we'll learn about recursive datatypes/functions in Haskell, using the simple example of propositional logic.

- Next week we'll talk about semantics, again using propositional logic as an example.
- For homework, read chapter 5 of (van Eijck, Jan and Unger, Christina, 2010) "Formal semantics for fragments".

Propositional logic

Sentences of propositional logic:

- $p \wedge q$
- $p \wedge (q \to r)$
- $p \vee \neg(q \wedge p)$
- $\bot \to (\top \vee (q \wedge r))$

To state a grammar for propositional logic, we first need a set of variables $\mathbf{Var} := \{ p, q, r, \ldots \}$, and a set of constants $\{ \top, \bot \}$.

- If $p \in \mathbf{Var} \cup \{ \top, \bot \}$, then p is a sentence of propositional logic.
- If ϕ, ψ are sentences of propositional logic, then $(\phi \wedge \psi)$ is a sentence of propositional logic.
- If ϕ, ψ are sentences of propositional logic, then $(\phi \vee \psi)$ is a sentence of propositional logic.
- If ϕ, ψ are sentences of propositional logic, then $(\phi \rightarrow \psi)$ is a sentence of propositional logic.
- If ϕ is a sentence of propositional logic, then so is $\neg\phi$

We'll use a *sum type* to implement the grammar of propositional logic. We can start with just the following:

```haskell
data PropL = PVar String deriving (Eq,Show)
```

```
ghci> :t (PVar "p1")
PropL
ghci> :t (PVar "p3")
PropL
```

Adding negation: recursive datatypes!

```
data PropL = PVar String | PNot PropL deriving (Eq,Show)
```

Note that we *reuse* `PropL` in the constructor for negative sentences; this means that if an expression `p` is of type `PropL`, then `PNot p` is of type `PropL`.

```
ghci> :t (PNot (PVar "p1"))
PropL
ghci> :t (PNot (PNot (PNot (PVar "p1")))
PropL
```

Note that this already gives us infinite inhabitants of type `PropL`.

Adding connectives

```
data PropL = PVar String | PNot PropL | PAnd PropL PropL | POr
↪  PropL PropL deriving (Eq,Show)
```

```
ghci> :t (PAnd (PVar "p1") (POr (PVar "p2") (PVar "p3")))
PropL
ghci> :t (PAnd (PVar "p1"))
error
```

Bonus: infix constructors!

Functions that take two arguments can be used as *infix operators* by enclosing them in backticks. This also goes for constructors:

```
ghci> :t PAnd
  PropL -> PropL -> PropL
ghci> :t ((PVar "p1") `PAnd` (PVar "p2"))
PropL
```

You can even use infix constructors in the data declaration:

```
data PropL = PVar String | PNot PropL | PropL `PAnd` PropL | PropL
 ↪  `POr` PropL deriving (Eq,Show)
```

Implementing a custom **Show** instance for **PropL** simply amounts to defining a function **show** of type **PropL -> String**.

```
data PropL = PVar String | PNot PropL | PropL `PAnd` PropL | PropL
 ↪  `POr` PropL deriving Eq

instance Show PropL where
  show (PVar s) = s
  show (PNot p) = "~" ++ show p
  show (p `PAnd` q) = "(" ++ show p ++ " & " ++ show q ++ ")"
  show (p `POr` q) = "(" ++ show p ++ " | " ++ show q ++ ")"
```

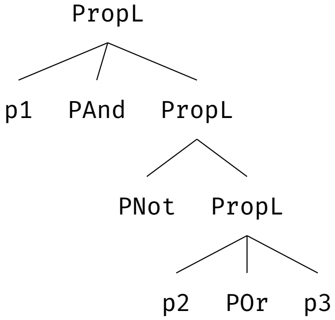
Custom `Show` cont.

The **`Show`** instance we just declared will automatically be used by ghci.

```
ghci> ((PVar "p1") `PAnd` (PNot ((PVar "p1") `POr` (PVar "p3")))
(p1 & ~(p2 | p3))
```

We can also use it explicitly by calling **`show`** on something of type **`PropL`**.

Recursive datatypes are used to create an Abstract Syntax Tree (AST) for sentences of propositional logic.



Let's say that we want to compute the number of operators in a formula.

In order to do so we'll need a recursive function `opsNr`.

First, we define the base of the recursion (where the recursion halts):

```
opsNr :: PropL -> Int
opsNr (PVar _) = 0
```

For all other cases we need recursion:

```
opsNr :: PropL -> Int
opsNr (PVar _) = 0
opsNr (PNot p) = 1 + opsNr p
opsNr (PAnd p q) = 1 + opsNr p + opsNr q
opsNr (POr p q) = 1 + opsNr p + opsNr q
```

Another recursive function: depth

```haskell
depth :: PropL -> Int
depth (PVar _) = 0
depth (PNot p) = 1 + depth p
opsNr (PAnd p q) = undefined
opsNr (POr p q) = undefined
```

Depth cont.

```haskell
depth :: PropL -> Int
depth (PVar _) = 0
depth (PNot p) = 1 + depth p
depth (PAnd p q) = 1 + max (depth p) (depth q)
depth (POr p q) = 1 + max (depth p) (depth q)
```

Exercise: gather names

- Exercise: write a recursive function that returns a list of all of the variables that occur in a formula.
- As a bonus, remove duplicates and sort the output alphabetically.

Normalization

- A formula of propositional logic is in *conjunctive normal form* (CNF) iff it is a conjunction of one of more clauses.
    - A clause is a disjunction of one or more literals; a literal is either a propositional variable, or the negation of a propositional variable.
    - Some formulas in conjunctive normal form:
- $(p \lor \neg q \lor r) \land s$
- $(p \lor q) \land r$
- p

Conversion to CNF and rules of inference

- Any formula can be converted into CNF by successively applying the following rules of inference:
  - $\neg\neg\phi \Rightarrow \phi$ (Double Negation Elimination; DNE)
  - $\neg(\phi \vee \psi) \Rightarrow \neg\phi \wedge \neg\psi$ (De Morgan's law 1; dM1)
  - $\neg(\phi \wedge \psi) \Rightarrow \neg\phi \vee \neg\psi$ (De Morgan's law 2; dM1)
  - $\phi \vee (\psi \wedge \rho) \Rightarrow (\phi \vee \psi) \wedge (\phi \vee \rho)$ (Distributive Law; DL)
- Conversion to CNF can be accomplished by:
  - Pushing negations in, by repeatedly applying dM.
  - Getting rid of any double negations via DNE.
  - Repeatedly applying DL, to get rid of disjunctions applying over conjunctions.

$$\neg(\neg(p \vee q) \wedge r) \wedge \neg(p \wedge r)$$

$$\begin{aligned}
&\Rightarrow (\neg\neg(p \vee q) \vee r) \wedge \neg(p \wedge r) \\
&\Rightarrow (\neg\neg(p \vee q) \vee r) \wedge (\neg p \vee \neg r) \\
&\Rightarrow (p \vee q \vee r) \wedge (\neg p \vee \neg r)
\end{aligned}$$

- Let's try to write a function to implement this procedure in Haskell.
- We know what the type of this function should be:

```
toCNF :: PropL -> PropL
toCNF = undefined
```

Pushing negation in

We'll break this function down into three steps.

- First, let's push negations inward by repeatedly applying dM.
- Second, let's get rid of any resulting double negations.
- Third, let's distribute conjunctions over disjunctions.

```
toCNF :: PropL -> PropL
toCNF = distributeConj . elimDN . pushNegsIn

dM :: PropL -> PropL
dM :: PropL -> PropL

dne :: PropL -> PropL
dne = undefined

distLaw :: PropL -> PropL
distLaw = undefined
```

First step: de Morgan’s

We can apply de Morgan’s via heavy use of *pattern matching*.

```
dM :: PropL -> PropL
dM (PNot (p `POr` q)) = (PNot p) `PAnd` (PNot q)
dM p = p
```

Pattern matching applies wherever possible, making the second line an elsewhere case.

de Morgan’s continued

```haskell
dM :: PropL -> PropL
dM (PNot (p `PAnd` q)) = (PNot p) `POr` (PNot q)
dM (PNot (p `POr` q)) = (PNot p) `PAnd` (PNot q)
dM p = p
```

- Note that the function we just defined is *not recursive*.
    - It only applies de Morgan's if the *top level formula* matches the structural description imposed by pattern matching.

```
ghci> dM (PVar "p1" `PAnd` PNot (PVar "p2" `POr` PVar "p3"))
(p1 & ~(p2 | p3))
```

- In order to eventually convert to CNF we need to apply `dM` *recursively*.
- Exercise: write a recursive variant of `dM`.

Recursive application of de Morgan's

```haskell
dM :: PropL -> PropL
dM (PNot (p `PAnd` q)) = (PNot (dM p)) `POr` (PNot (dM q))
dM (PNot (p `POr` q)) = (PNot (dM p)) `PAnd` (PNot (dM q))
dM (PNot p) = PNot (dM p)
dM (p `PAnd` q) = dM p `PAnd` dM q
dM (p `POr` q) = dM p `POr` dM q
dM (PVar p) = PVar p
```

- Double negation elimination is a bit simpler.
- First, the non-recursive variant:

```
dne :: PropL -> PropL
dne (PNot (PNot p)) = p
dne p = p
```

- Exercise: make this apply recursively

Recursive DNE

```haskell
dne :: PropL -> PropL
dne (PNot (PNot p)) = dne p
dne (PNot p) = PNot (dne p)
dne (p `PAnd` q) = dne p `PAnd` dne q
dne (p `POr` q) = dne p `POr` dne q
dne (PVar p) = PVar p
```

Distributive Law

First, the non-recursive variant (we're really stretching the limits of pattern matching):

Our statement of the distributive law actually subsumes three different cases:

```
distLaw :: PropL -> PropL
distLaw ((p `PAnd` q) `POr` (r `PAnd` s)) = (p `POr` r) `PAnd` (p `POr` s) `PAnd` (q `POr` r) `PAnd` (q `POr` s) -- double distributivity
distLaw (p `POr` (q `PAnd` r)) = (p `POr` q) `PAnd` (p `POr` r) --left dist
distLaw ((q `PAnd` r) `POr` p) = (q `POr` p) `PAnd` (r `POr` p) --right dist
distLaw p = p
```

- Exercise: write the recursive variant!

Recursive distributive law

```
distLaw :: PropL -> PropL
distLaw ((p `PAnd` q) `POr` (r `PAnd` s)) = (distLaw p `POr`
↪  distLaw r) `PAnd` (distLaw p `POr` distLaw s) `PAnd` (distLaw
↪  q `POr` distLaw r) `PAnd` (distLaw q `POr` distLaw s) --
↪  double distributivity
distLaw (p `POr` (q `PAnd` r)) = (distLaw p `POr` distLaw q)
↪  `PAnd` (distLaw p `POr` r) --left dist
distLaw ((q `PAnd` r) `POr` p) = (distLaw q `POr` distLaw p)
↪  `PAnd` (distLaw r `POr` distLaw p) --right dist
distLaw (PNot p) = PNot (distLaw p)
distLaw (p `PAnd` q) = distLaw p `PAnd` distLaw q
distLaw (p `POr` q) = distLaw p `POr` distLaw q
distLaw (PVar p) = PVar p
```

- A function that maps formulas of propositional logic to a truth table.
- You can treat a truth table as a list of *pairs* of variable assignments and truth values.
- A variable assignment is a list of pairs of variables and truth values.

$\mathcal{F}in$

van Eijck, Jan and Unger, Christina (2010). *Computational Semantics with Functional Programming*, Cambridge University Press.