# Fun with Lattices

Jose Cardona

OQuant

LambdaConf 2019

# Why Haskell?

Haskell is:

- Pure! (Never worry about side effects again!)
- Declarative (wholemeal programming).
- Likely older than you! (And me!)
- Statically typed, with type inference.
- Functional! (unlike my attention span)

# Why not haskell

For the most part, the reasons "not to use haskell" are similar to the reasons you'd avoid GC'ed languages in the first place:

- You're building a piece of software that has resource constraints (RTOS, AAA Games).
- You're building a piece of software that has incredibly tight latency requirements (HFT).
- You're not a fun person (but we can fix that, you're here! that's the first step)

# A lil' haskell

A simple declaration: We read this as "x has type Int"

```
x :: Int
x = 5
```

Haskell comes equipped with a few basic types built-in:

- Int = Machine-dependent sized integer
- Integer = Arbitrary precision integer (Java folks will know it as bigInt)
- Double, Float = Same as other languages (32/64 bit precision floats)
- Lists (written as $[a]$ where a is some type)
- n-ary tuples (a, b), (a, b, c), etc.

# A lil' more haskell

Let's try every programmer's favorite definition to start a language: the factorial function.

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

You might notice a few peculiar things:

- There's "multiple lines" of the factorial function.
- somehow there's a literal 0 on the left hand side.

The ability to "inspect the structure" of an argument is a particular concept called "pattern matching" which is central to the way you write haskell code. It's very similar to writing structurally inductive proofs.

Recall well formed formulae. An expression $A$ is a well formed formula if it is of the following form:

- it is an "Atomic" metavariable $p, q, r, s...$
- It is of the form $(\neg A)$ where A is a WFF
- It is of the form $A \circ B$ where A and B and WFF and $\circ$ is a logical connective

Proving $((\neg p) \vee q) \wedge q$ is a WFF simply involves recursively traversing the statements (top down or bottom up) and building the WFF that way. Pattern matching works on that same premise.

# Arithmetic Operators

- $+, -, *$ work as you expect, however: Haskell *does not* have any sort of implicit numeric conversions. You may come to appreciate this later, but: to go from a fractional type to an integral type: use *round*, *ceiling*, *floor*. To go from an integral to a rational, use *fromIntegral*.
- modulo is *mod* (not %).
- / is for fractional division. For integral division, use *div*

# Booleans

- *True* and *False*.
- logical negation is just *not*. It's just a function.
- &&, || logical operators
- For ordering/equality: use $==, >, >=, <, <=$ though note: equality is structural equality in haskell, and comparison operators only operate on the same type.

# Algebraic data types

if you're familiar with enums in java:

```
public enum Level {
 HIGH,
 LOW,
 EVEN_LOWER,
 INFO,
 SWEET_N_LO
}
```

We have the same in haskell:

```
data Level = High | Low | EvenLower | Info | SweetNLo
```

Similar to enums: the compiler assists us quite a bit in determining if we're missing a case for our enums if we don't provide it. However, there's some power algebraic data types have that enums do not

# Algebraic Data types

- Algebraic data types can contain data.
- Algebraic data types can be recursive.
- Algebraic data types can be pattern matched on.

```
data ArithExpr
  = Add ArithExpr ArithExpr
  | Sub ArithExpr ArithExpr
  | Value Int
```

Example, with the above defined, we can now sum on this tree:

```
eval :: ArithExpr -> Int
eval (Add l r) = eval l + eval r
eval (Sub l r) = eval l - eval r
eval (Value i) = i
```