Haskell and Category Theory

Structure

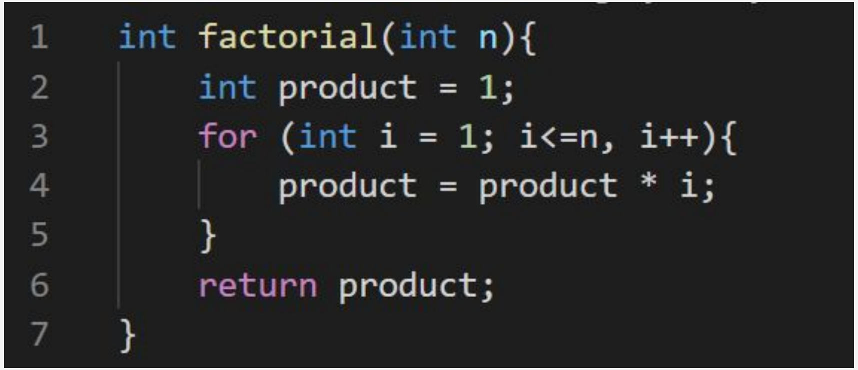
- Introduce Haskell
  - Contextualize
  - Language basics
- Haskell’s type system
  - Category **Hask**
- Currying
- Language-level categorical constructs
  - Functors
  - Monoids
  - Monads

Haskell: Context

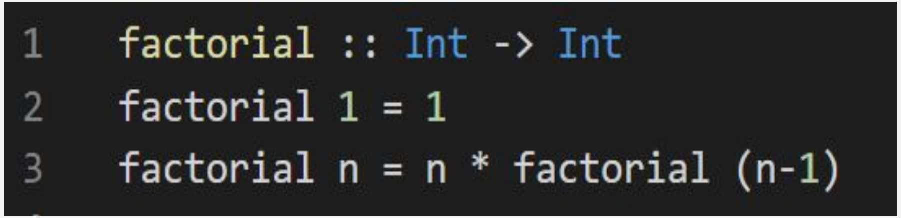
- “Conventional” program structure: Imperative
  - Java, C, Python…
  - Describe “how” program does something
  - program is series of steps (control flow)
    - For loops, if/then...

- Alternative: Declarative
  - Functional languages typically declarative
    - Haskell
  - Describe logic but *don’t* describe control flow
    - Functions, recursion...

Examples: Factorial



```
int factorial(int n){
    int product = 1;
    for (int i = 1; i<=n, i++){
        product = product * i;
    }
    return product;
}
```



```
factorial :: Int -> Int
factorial 1 = 1
factorial n = n * factorial (n-1)
```

Haskell

- Declarative, functional language
  - “what” not “how”
  - Programs are collections of functions not sequences of steps
    - Higher-orderism: functions passed around as parameters/results
- Built from lambda calculus
  - Type theoretic system for specifying computation
- Known for mathematical formalism in underlying structure/language tools
  - Built *with* category theory, not *for* category theory
- Syntax notes:
  - Function application via space: *f r* not *f(r)*
  - Composition via periods: `(f . g) x = f (g x)`

Typing in Haskell - a quick, pragmatic view

Data

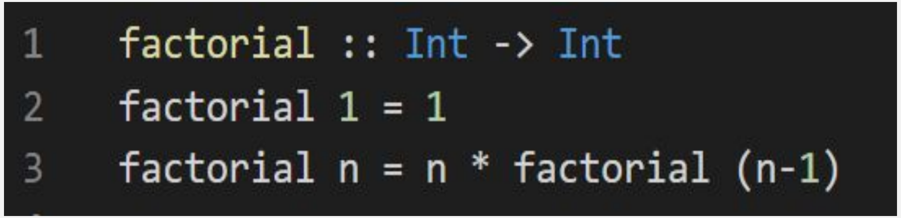
- Data type: set of values
    - Int: [-2^29...2^29 - 1]
    - String: [a-zA-Z…]*

Functions

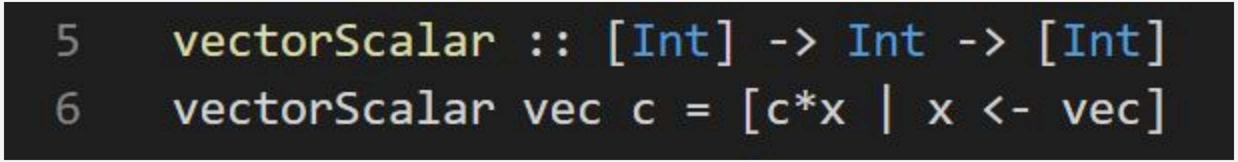
- specifies types of function inputs/output (data)
    - Int -> String takes int, returns string
- typechecking: crucial for writing correct software
    - (f . g) x:
        - is x g’s input type?
        - is g’s result type f’s input type?

Example type signatures

- Function typing: types of parameters, result



```haskell
factorial :: Int -> Int
factorial 1 = 1
factorial n = n * factorial (n-1)
```



```haskell
vectorScalar :: [Int] -> Int -> [Int]
vectorScalar vec c = [c*x | x <- vec]
```

Typing cont.

- Typeclasses
  - Groups of types that define specific behavior
  - ex: types in Eq typeclass have to support ‘==’ function
    - tests equality
  - types in Ord typeclass have to support ordinal comparisons
    - <, >, etc
  - Int? String?

- Algebraic data types
  - User created
  - associated with powerfully abstract type “groups” via typeclasses

- Type variables (generic types)
  - functions that don’t require specific types use variables in place

Example: ==

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
Prelude>
```

Hask

Hask

- Categorical representation of Haskell’s type system
- Ob(**Hask**): Haskell types (Int, String, [Int]...)
    - Don’t care about values!
    - Int -> Int not 2 -> 3
- **Hask**(A, B): functions A -> B
    - Extensionally identified
        - I/O pairs same = same function
- Morphism composition: function composition
    - `f . g  = \x -> f ( g x )`

Identities

Identity morphism for A ∈ **Hask**:

f :: A -> A

True or false: Since **Hask** doesn't care about values, only types, any function A -> A can be interpreted as A's identity morphism in **Hask**.

False - composition laws violated. Counterexample:

Let A = Int. Consider (+1) :: Int -> Int, (*2) :: Int -> Int.

If (+1) can be interpreted as the identity, (*2) . (+1) = (+1) . (*2) = (*2)

id

- Identity morphisms in **Hask: id** function

id :: A -> A

id x = x

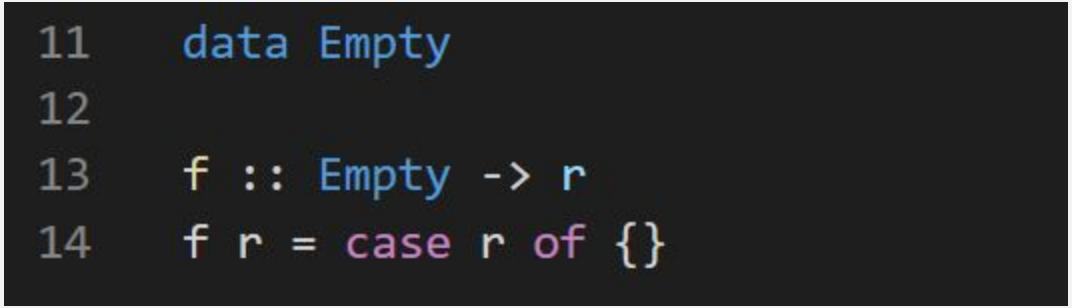
- Parametric polymorphism
    - Type variable ‘A’ instead of concrete type: can be any element of Ob(**Hask**)
    - Too general to serve as identity morphism
- id instantiated with a concrete type (ex. Int -> Int) serves as identity morphism

“Platonic” **Hask**

- Implementation level details break underlying categorical structure
- “Platonic” **Hask**:
  - Category corresponding to subset of haskell
  - Types don’t have “bottom” values
    - anything that makes a program state undefined: non-terminating loops, exceptions…
    - ‘Undefined’
    - Lazy evaluation
- Removes implementation problems
  - gives **Hask** expected attributes/structures
    - Initial objects, terminal objects, products, coproducts
  - makes categoric features behave as their names suggest
    - Functor, Monad typeclasses

Hask: Initial objects

Requirement: $a \in$ **Hask** s.t. $\forall b \in$ **Hask**, $\exists! f :: a \to b$



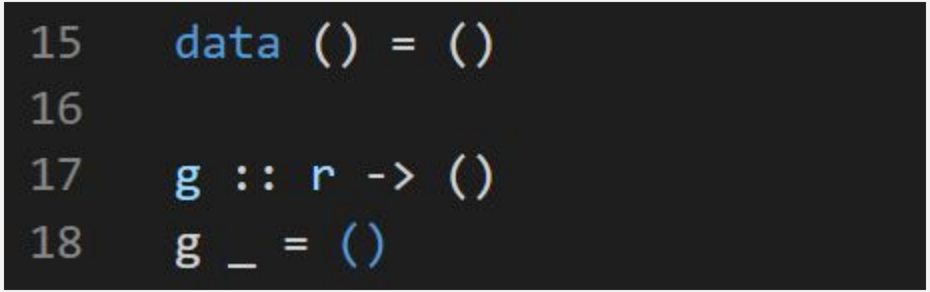
```haskell
data Empty

f :: Empty -> r
f r = case r of {}
```

- Real **Hask**: Empty type can be “undefined” (a bottom value)

Hask: Terminal objects

Requirement: $a \in$ **Hask** s.t. $\forall b \in$ **Hask**, $\exists!\ g :: b \to a$

- (): unit type
    - both a type and a value
    - analogue of singleton set



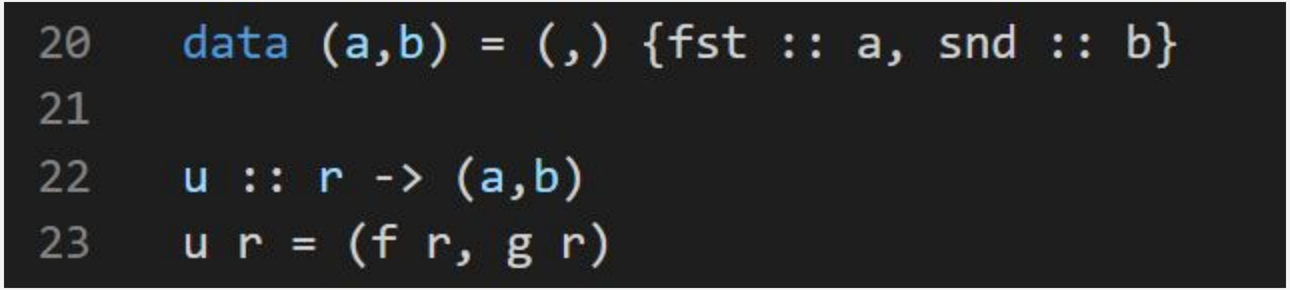
```haskell
data () = ()

g :: r -> ()
g _ = ()
```

- again assuming no problems from “undefined”

Hask: Products

Requirement: $\forall f :: r -> a, g :: r -> b$

$\exists!\ u :: r -> product(a, b)$ s.t $\pi_1 . u = f, \pi_2 . u = g$



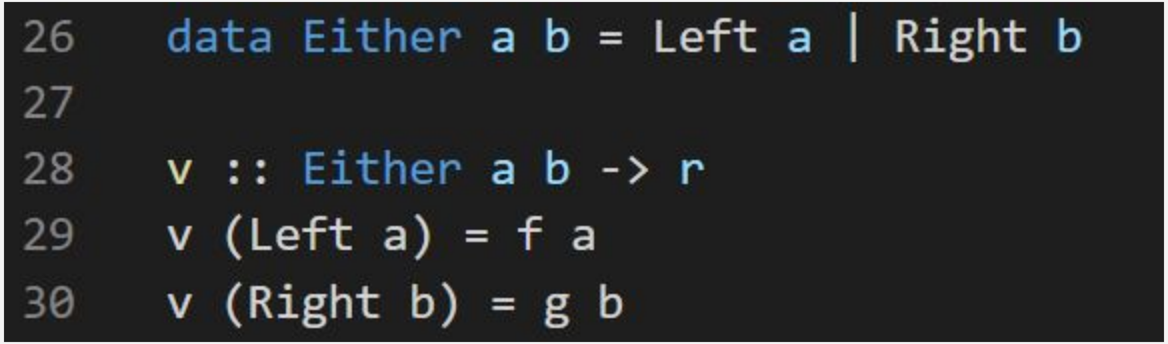
```haskell
data (a,b) = (,) {fst :: a, snd :: b}

u :: r -> (a,b)
u r = (f r, g r)
```

- $\pi_1 = fst :: (a, b) -> a$
- $\pi_2 = snd :: (a, b) -> b$

Hask: Coproducts

Requirement: $\forall f :: a -> r, g :: b -> r,$

$\exists!\ v :: coproduct(a,b) -> r$ s.t $(v . i_1) = f, (v . i_2) = g$



```haskell
data Either a b = Left a | Right b

v :: Either a b -> r
v (Left a) = f a
v (Right b) = g b
```

Hask: summary

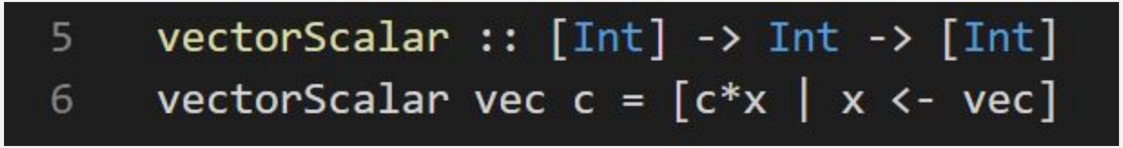
- Categoric representation of Haskell’s type system
  - Ob(**Hask**): types
  - Morphisms: functions between types
- (Platonic) **Hask** is Cartesian closed
  - ‘Undefined’ and other misbehaving constructs removed
  - See online resources for more discussion

	Initial object	Terminal object	Products	Coproducts
Hask	✗	✗	✗	✗
Platonic **Hask**	data Empty	data () = ()	data (a,b) = ...	data Either a b = Left a | Right b

Currying

Currying

- Haskell functions all take only one parameter under the hood
- We’ve seen multi-parameter functions:

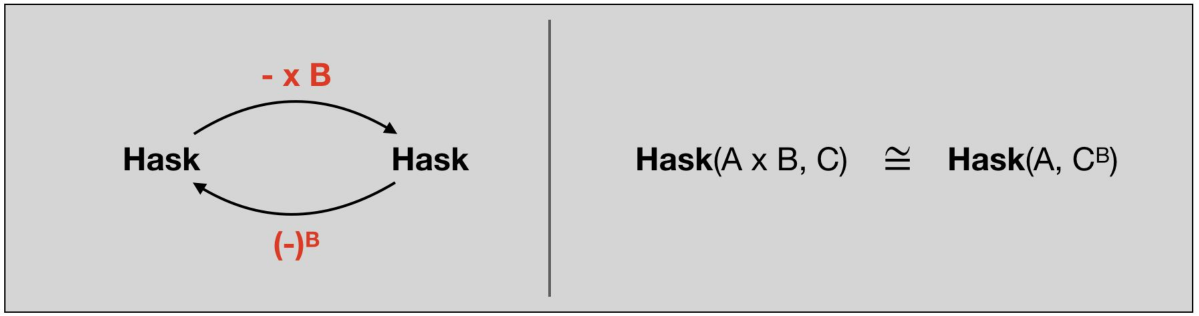


```
vectorScalar :: [Int] -> Int -> [Int]
vectorScalar vec c = [c*x | x <- vec]
```

- Currying: a clever trick
  - *n*-ary function takes one parameter and returns an *(n-1)*-ary function
  - vectorScalar :: [Int] -> (Int -> [Int])
- Partial application: feeding *a* parameters to *n*-ary function returns *(n-a)*-ary function
  - Create functions on the fly
  - Generically defined top-level functions + partial application implicitly specifies huge range of functions
- Example: ‘+’
  - + :: Int -> Int -> Int
    - + 2 3 = 5
  - + 2 :: Int -> Int
    - (+2) 3 = 5

Currying: a categorical relationship

- All *n*-ary functions can be represented as chained 1-ary functions
    - Category theory connection?
- Adjunction
    - let *A, B, C* ∈ **Hask**



$$\begin{tikzcd} \mathbf{Hask} \arrow[r, bend left, "- \times B"] & \mathbf{Hask} \arrow[l, bend left, "(-)^B"] \end{tikzcd}$$

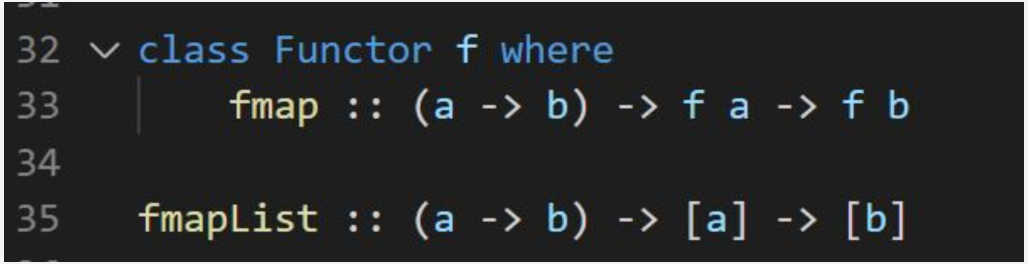
$$\mathbf{Hask}(A \times B, C) \cong \mathbf{Hask}(A, C^B)$$

- Exponential objects: function types are types too
- Left adjoint: product functor, right adjoint: exponentiation functor

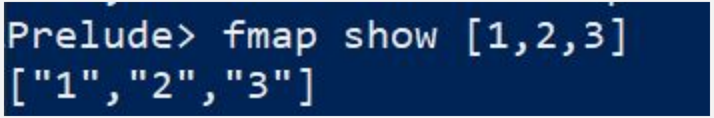
Categoric Typeclasses

Functor

- Functor: typeclass (group of types)
  - a Functor is a container type that can be mapped over
    - list, tree…



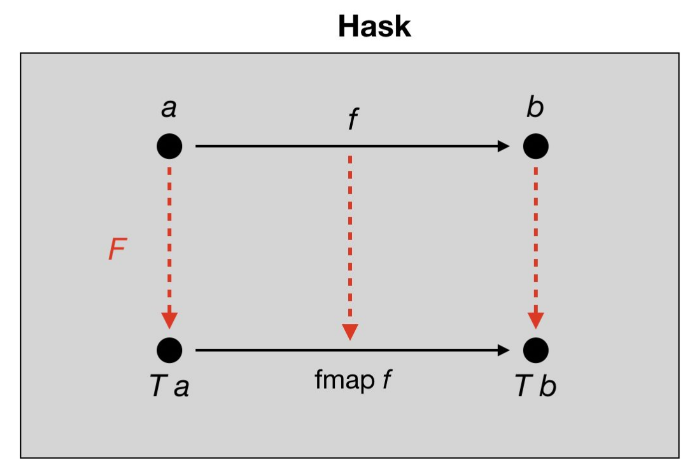
```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b

fmapList :: (a -> b) -> [a] -> [b]
```



```
Prelude> fmap show [1,2,3]
["1","2","3"]
```

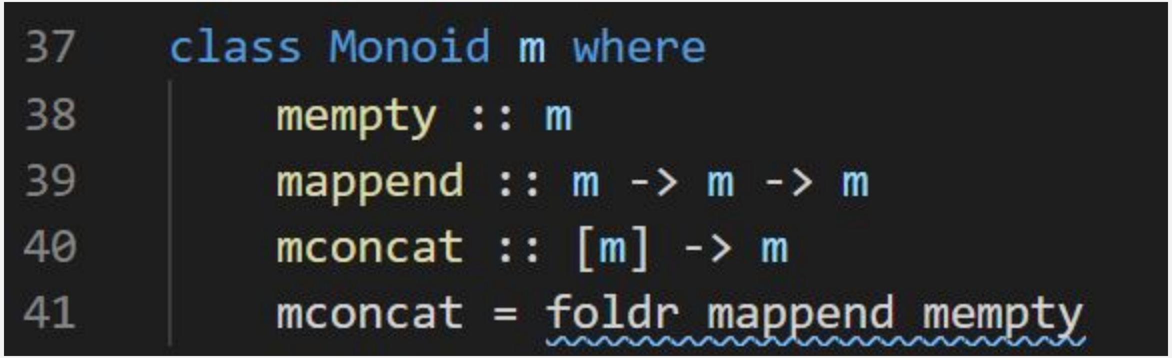
The Functor typeclass in **Hask**

- Instance T of Functor: endofunctor F on **Hask**
    - for $a \in$ **Hask**, $F\ a = T\ a$
    - For $f : a \to b$, $F\ f : T\ a \to T\ b$



Monoid

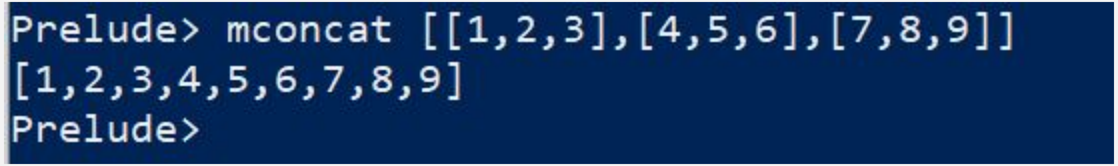
- **Hask**: types and functions between types
- Structure within a type?
- Monoid typeclass
  - set with unit and associative binary operation



```haskell
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

Monoid example

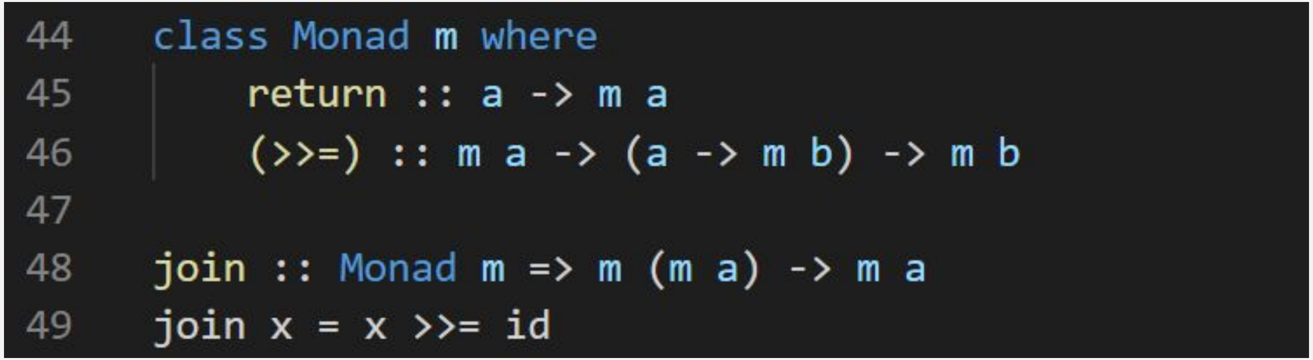
- simple monoid: List
  - **Unit?**
    - mempty = []
  - **Binary op?**
    - mappend a b = a ++ b



```
Prelude> mconcat [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,4,5,6,7,8,9]
Prelude>
```

Monads (the sparknotes)

- Category theoretic monad: triple (T, η, μ)
    - $T: C \longrightarrow C$ (functor)
    - $\eta : 1_C \longrightarrow T$ (n. t.)
    - $\mu : T^2 \longrightarrow T$ (n. t.)



```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b

join :: Monad m => m (m a) -> m a
join x = x >>= id
```

- endofunctor T is m (C is **Hask**)
- η is *return*
- μ is *join*

Summary

- Haskell is an extremely elegant programming language
  - Design guided by category theory
  - Language-level constructs leverage powerful mathematical abstractions
- Most notable language heavily adopting PL theory -> category theory connection
  - Type system: **Hask**
  - Currying adjunction
  - Categoric typeclasses (Functor, Monad…)
- Resources
  - Learning Haskell
    - GHC - Glasgow Haskell Compiler
    - Learn You A Haskell
  - Category theory in Haskell
    - Bartosz Milewski’s blog
    - Course website

Controversy

- “Hask is not a category” - Andrej Bauer
    - Effectively describes how aspects of Haskell break the underlying categoric model
    - “People walk away from Haskell thinking they know some category theory where in fact they have not even seen a category yet”
    - “[I am objecting to] The fact that some people find it acceptable to defend broken mathematics on the grounds that it is useful. Non-broken mathematics is also useful, as well as correct. Good engineers do not rationalize broken math by saying “life is tough”.”
- Is this relevant?
    - Haskell already notorious in CS world for being overly academic
        - CS breaks nice mathematical abstractions all the time
    - Understanding why **Hask** is not a category (*seq*, bottom values) takes more understanding of category theory than most Haskellers have/than is required for the abstraction to provide valuable insight/structure/rigor to their programming
    - “Category theory is a powerful enough substrate that even doing it wrongly adds a lot of utility” -Edward Kmett

Questions