Haskell: Programming with Functions

Overview of Day 3

Niki Vazou, Pablo Serrano
Clubes de Ciencia, Summer 2015

Cabal

the Haskell package manager

Type Classes

A type class is an interface that defines some behavior.

class name

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

class method(s)

Type Class Instances

Define how a type instantiates a class

```
instance Eq a => Eq (Err a) where
 err1 == err2 = eqErr err1 err2
```

Now I can use (==) on `Err a` expressions

Type Class Constraints

Constrain type variables to be instances of type class

constraint

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

Constraints are propagated!

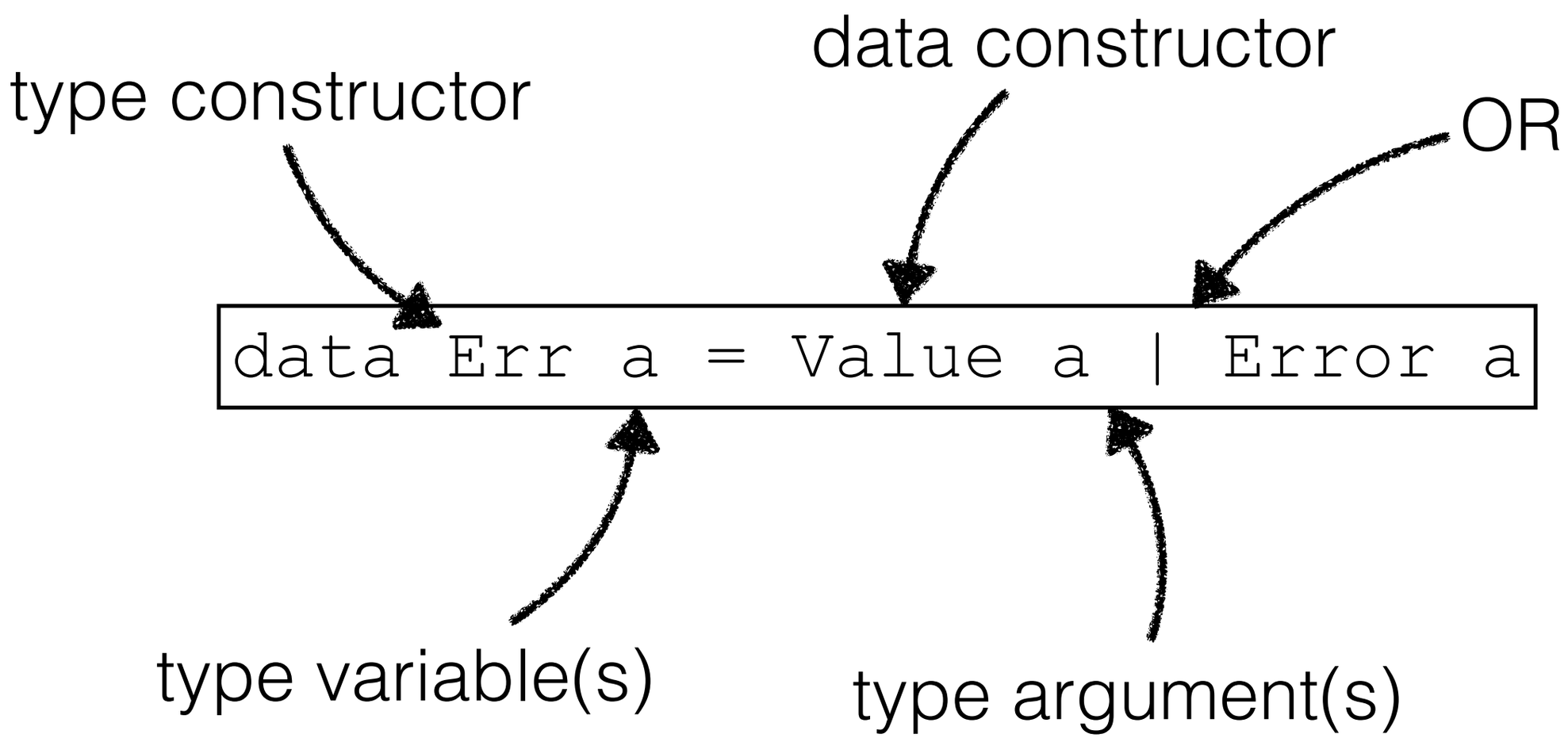
```
checkEq :: (Eq a) => a -> a -> String
checkEq x y | x == y = “Equal”
            | otherwise = “Different”
```

Why Type Classes?

- It is an abstraction!
- Use the *same* (==) operator to things of different type
- Determine the implementation of (==) from the type
- Haskell will figure out the correct implementation
- Overloading

Data Types

Usually, a collection of labelled things



List Manipulation

Recursion

```
length []     = 0
length (x:xs) = 1 + length xs
```

Higher Order Functions

```
map :: (a -> b) -> a -> b
```

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

Structural Induction

```
length []     = 0
length (x:xs) = 1 + length xs
```

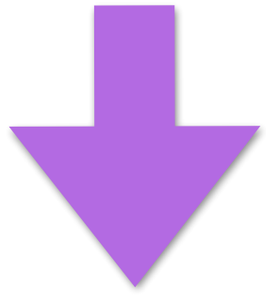
Prove that for every list xs, length xs >= 0

If S([]) and S(xs) => S(x:xs), then for all ls. S(ls)

Currying

Transform a function with many arguments
to a function with one argument

```
plus1 :: (Int, Int) -> Int
plus1(x, y) = x + y
```



```
plus2 :: Int -> Int -> Int
plus2 x y = x + y
```

If we apply the first argument we get a function

List Comprehension

```
[x | x <- xs, y <- ys, p x, q y]
```

Multiplication Tables

```
[(7, 7*i)| i <- [1..10]]
```

All Combinations

```
[(i,j)| i <- [1..10], j <- [1..i]]
```

Today

Binary Search Trees

String Manipulation

Monads

Type Inference: The theory break