CSCI-UA 490

Type Classes

Edward Z. Yang

The Problem

```
member :: a -> [a] -> Bool
member x []                    = False
member x (y:ys) | x==y         = True
                | otherwise    = member x ys
```

Does this really work for any type a? What about functions?

Similar Problems

```
sort :: [a] -> [a]
(+) :: a -> a -> a
show :: a -> String
serialise :: a -> ByteString
hash :: a -> Int
```

Non-solution: Local choice

[Standard ML]

$$\begin{tikzcd}
 & a \text{ `multInt` } b \\
a * b \arrow[ur, squiggly] \arrow[dr, squiggly, "\text{desugars to}"'] & \\
 & a \text{ `multFloat` } b
\end{tikzcd}$$

monomorphic

```
square x = x*x :: Int
```

3 * 3 ✓

3.14 * 3.14 ✓

square 3 ✓

square 3.14 ✗ Problem!

Non-solution: Local choice (v2)

[C++]

$$\begin{tikzcd}
 & \text{square } x = x \text{ `multInt` } x \\
\text{square } x = x * x \arrow[ur, squiggly] \arrow[dr, squiggly, "\text{defines}"'] & \\
 & \text{square } x = x \text{ `multFloat` } x
\end{tikzcd}$$

Problem:

```
square x y = (square x, square y)
```

Exponential code blow up!

Non-solution: Provide it for everything

```
(==) :: a -> a -> Bool   <- Really!
```

```
3 * 3 == 9                      => True
(\x -> x) == (\x -> x+1)        => Runtime error
```

Problems: Not extensible
Runtime errors
Abstraction violating

Non-solution: "eqtype" polymorphism

$(==) :: a_{(==)} \to a_{(==)} \to \text{Bool}$

special type variable restricted to types with equality

$\text{member} :: a_{(==)} \to [a_{(==)}] \to \text{Bool}$

Problems: What about everything else?

Type classes

Works for any type 'a', provided 'a' is of type class Num.

```
square :: Num a => a -> a
square x = x * x
```

Similar:

```
sort :: Ord a => [a] -> [a]
serialize :: Show a => a -> String
member :: Eq a => a -> [a] -> Bool
```

(GHCi here)

Type classes

Works for any

```
square :: Num n => n -> n
square x = x * x
```

forget all you know about OO classes!

```
class Num a where
    (+) :: a -> a -> a
    (*) :: a -> a -> a
    ...
```

Class declaration

what are the Num operations?

```
instance Num Int where
    a + b = plusInt a b
    a * b = mulInt a b
    ...
```

instance declaration

how are the Num operations implemented for the type

How type classes work

```
square :: Num n => n -> n
square x = x * x
```

$\Rightarrow$

```
square :: Num n -> n -> n
square d x = (*) d x x
```

an extra <u>value</u> argument, of data type Num n

A value of type Num T is a vector of Num operations for T

How type classes work

```
square :: Num n => n -> n
square x = x * x
```

$\Rightarrow$

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  ...etc...
```

$\Rightarrow$

DATA TYPE DECLARATION:

```
data Num a
   = MkNum (a -> a -> a)
           (a -> a -> a)
           (a -> a)
           ...etc...
```

SELECTOR FUNCTION:

```
(*) :: Num a -> a -> a -> a
(*) (MkNum _ m _ ...) = m
```

QuickCheck

$$\begin{tikzcd}
 & \text{Generate random data} \\
\text{Type classes} \arrow[ur, bend left] \arrow[dr, bend right] & \\
 & \text{Testing functions}
\end{tikzcd}$$

- Arbitrary Int

- Testable Bool
- (Arbitrary a, Testable b) $\Rightarrow$ Testable (a $\rightarrow$ b)

Type classes versus OO

```
class Show a where
    show :: a -> String
```

$\overset{?}{\approx}$

```
interface Show {
    String show();
}
```

Type classes versus OO

```
class Show a where
    show :: a -> String
```

No!

$\cancel{\approx}$

```
interface Show {
    String show();
}
```

type-based dispatch

value-based dispatch

Type classes versus OO

```
read2 :: (Read a, Num a) => String -> a
read2 s = read s + 2
```

$\Downarrow$

```
read2 dr dn s = (+) dn (read dr s)
                       (fromInteger dn 2)
```

↑ ↑

dictionaries in,
value out!

Type classes versus OO interfaces

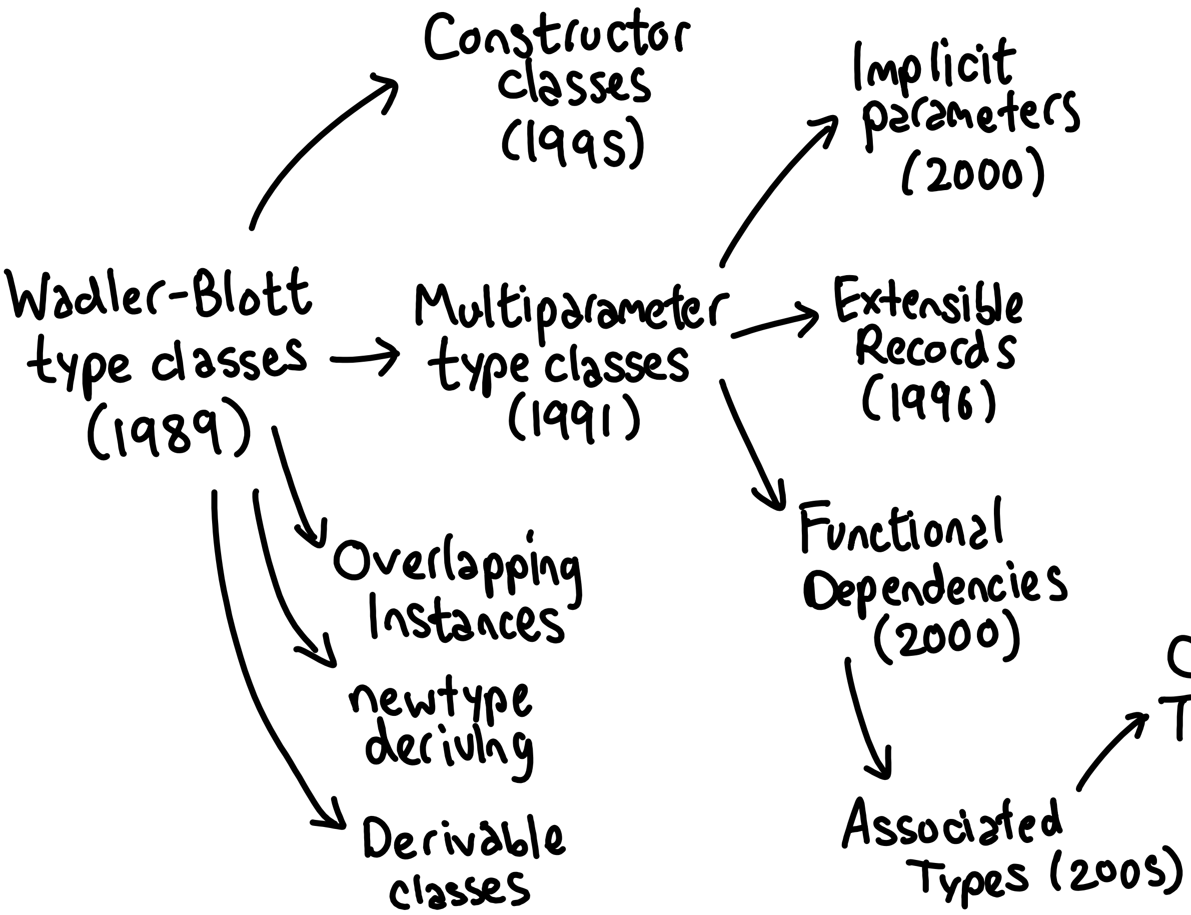
- Multiple constraints easy
  (can do in Java w/ F-bounded quantification)
- Can retroactively give instances to types

```
class Wibble a where
   wibble :: a → Bool

instance Wibble Int where
   wibble x = x==1
```

- Haskell has no subtyping → binary methods, no variance
  more on this later!

Type classes over time



Wadler-Blott type classes (1989)
Constructor classes (1995)
Multiparameter type classes (1991)
Implicit parameters (2000)
Extensible Records (1996)
Functional Dependencies (2000)
Associated Types (2005)
Closed Type Families (2013)
Overlapping Instances
newtype deriving
Derivable classes

Applications

Type-level computation

Generic programming

Testing

Type classes: "The most unusual feature of Haskell's type system."

- more flexible than originally realized
- plethora of research topics
- big influence on new languages
  (Rust traits, C++ concepts, ...)