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]

a * b

desugars to

a `multInt` b

a `multFloat` b

monomorphic

square x = x * x :: Int

3 * 3 ✓

3.14 * 3.14 ✓

square 3 ✓

square 3.14 ✗ Problem!

Non-solution: Local choice (v2) [C++]

square x = x * x $\xrightarrow{\text{defines}}$

square x = x `multInt` x

square x = x `multFloat` x

Problem:

square x y = (square x, square y)

Exponential code blow up!

Non-solution: Provide it for everything

$(==) :: a \to a \to Bool$ ← Really!

$3 * 3 == 9 \implies$ True

$(\backslash x \to x) == (\backslash x \to x+1) \implies$ Runtime error

Problems: Not extensible
Runtime errors
Abstraction violating

Non-solution: "eqtype" polymorphism

$$(==) :: a(==) \to a(==) \to \mathrm{Bool}$$

special type variable restricted to types with equality

$$\mathrm{member} :: a(==) \to [a(==)] \to \mathrm{Bool}$$

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 declaration

what are the Num operations?

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

instance declaration

how are the Num operations implemented for the type

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

How type classes work

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

⇒

```
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
```

```
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...
```

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

Type classes

Generate random data

- Arbitrary Int

Testing functions

- Testable Bool
- (Arbitrary a, Testable b) ⇒ Testable (a → b)

Type inference

constraint set

$$f\ x = x + 2 == 3 \implies (Eq\ a,\ Num\ a) \Rightarrow a \to Bool$$

ordinary Hindley-Milner type

Type inference

$$f\ x = x + 2 == 3 \Rightarrow (Eq\ a,\ Num\ a) \Rightarrow a \to Bool$$

Type inference

$$f\ x = x + 2 == 3$$

$$(\mathrm{Eq}\ a,\ \mathrm{Num}\ a) \Rightarrow a \to \mathrm{Bool}$$

fromInteger

Constraint sets

1. Generate constraints
2. Simplify constraints

1. Generate constraints

constraints

$\{ Eq\ \tau_1 \}$

$\tau_1 = \tau_2$

$f\ x\ y = x == y$

τ_1 τ_2

type equalities

2. Simplify constraints

$$\{\text{Num } a, \text{Num } a\} \Rightarrow \{\text{Num } a\}$$

$$\{\text{Eq } [a], \text{Eq } a\} \Rightarrow \{\text{Eq } a\}$$

if instance $\text{Eq } a \Rightarrow \text{Eq } [a]$

$$\{\text{Eq } a, \text{Ord } a\} \Rightarrow \{\text{Ord } a\}$$

if class $\text{Eq } a \Rightarrow \text{Ord } a$

e.g. $\{\text{Eq } a, \text{Eq } [a], \text{Ord } a\} \Rightarrow \{\text{Ord } a\}$

Type classes versus OO

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

$\overset{?}{\approx}$

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

Type classes versus OO

No!

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

$\not\approx$

```
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
```

```
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

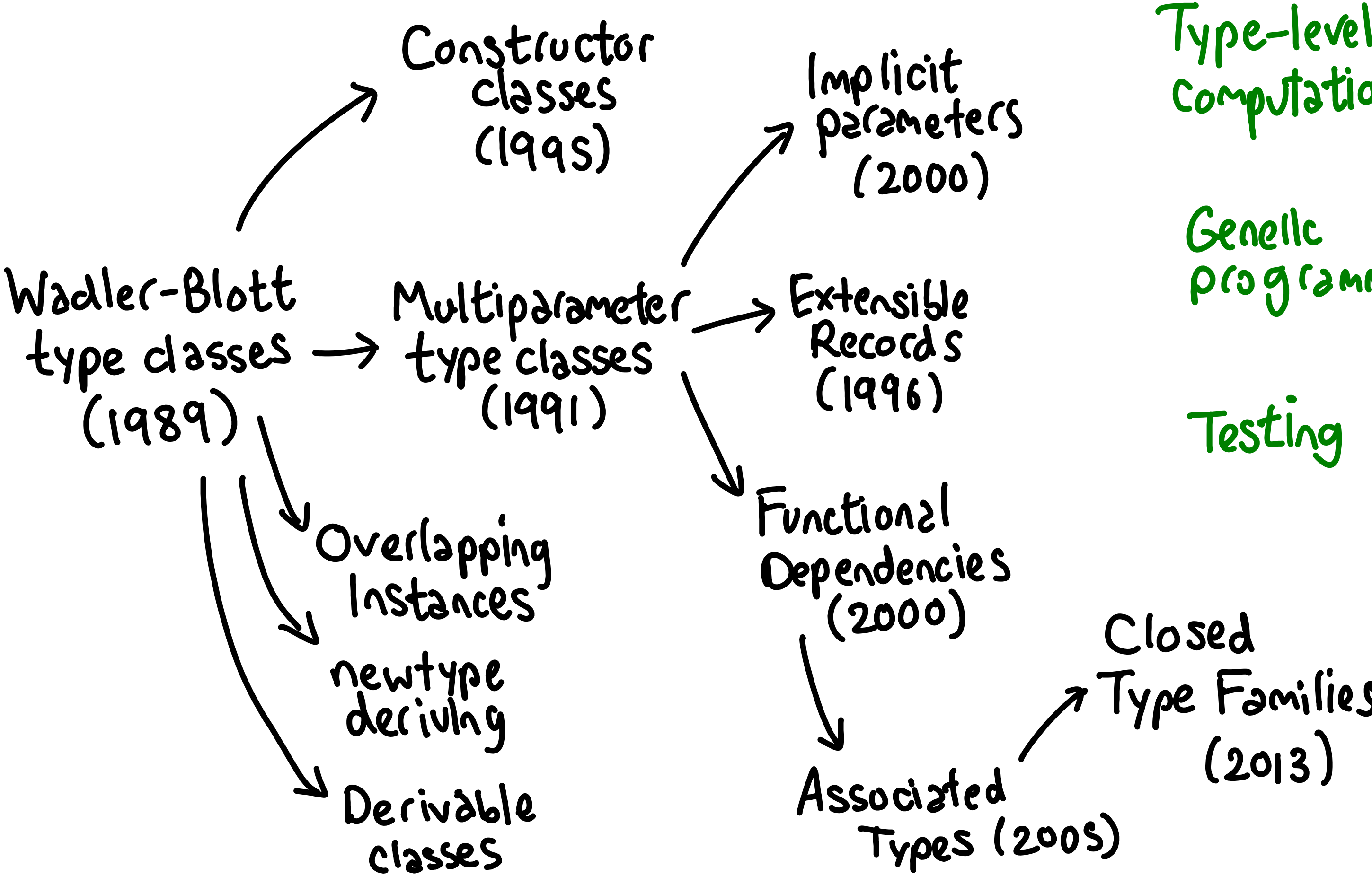
```
class Wibble a where
  wibble :: a -> Bool

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

- Haskell has no subtyping
  more on this later!

binary methods
no variance

Type classes over time



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, ...)