cs242

TYPE CLASSES

Kathleen Fisher

Reading: "A history of Haskell: Being lazy with class",
Section 3 (**skip** 3.9), Section 6 (skip 6.4 and 6.7)
"How to Make Ad Hoc Polymorphism less ad hoc",
Sections 1 – 7
"Real World Haskell", Chapter 6: Using Typeclasses

Thanks to Simon Peyton Jones for some of these slides.

Polymorphism vs Overloading

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by *any* type
  - if `f:t→t` then `f:int→int`, `f:bool→bool`, ...
- Overloading
  - A single symbol may refer to more than one algorithm
  - Each algorithm may have different type
  - Choice of algorithm determined by type context
  - Types of symbol may be arbitrarily different
  - + has types `int*int→int`, `real*real→real`, *but no others*

Why Overloading?

Many useful functions are not parametric.

- Can member work for any type?

```
member :: [w] -> w -> Bool
```

No! Only for types `w` for that support equality.

- Can sort work for any type?

```
sort :: [w] -> [w]
```

No! Only for types `w` that support ordering.

Why Overloading?

Many useful functions are not parametric.

- Can serialize work for any type?

```
serialize:: w -> String
```

No! Only for types `w` that support serialization.

- Can sumOfSquares work for any type?

```
sumOfSquares:: [w] -> w
```

No! Only for types that support numeric operations.

Overloading Arithmetic

First Approach

- Allow functions containing overloaded symbols to define multiple functions:

```
square x = x * x          -- legal
-- Defines two versions:
-- Int -> Int and Float -> Float
```

- But consider:

```
squares (x,y,z) =
    (square x, square y, square z)
-- There are 8 possible versions!
```

- This approach has not been widely used because of exponential growth in number of versions.

Overloading Arithmetic

Second Approach

- Basic operations such as + and * can be overloaded, but not functions defined in terms of them.

```
3 * 3              -- legal
3.14 * 3.14        -- legal

square x = x * x   -- int -> int
square 3           -- legal
square 3.14        -- illegal
```

- Standard ML uses this approach.
- Not satisfactory: Why should the language be able to define overloaded operations, but not the programmer?

Overloading Equality

First Approach

- Equality defined only for types that *admit equality*: types not containing function or abstract types.

```
3 * 3 == 9             -- legal
'a' == 'b'             -- legal
\x->x == \y->y+1       -- illegal
```

- Overload equality like arithmetic ops + and * in SML.
- But then we can't define functions using `==`:

```
member [] y     = False
member (x:xs) y = (x==y) || member xs y

member [1,2,3] 3        -- illegal
member "Haskell" 'k'    -- illegal
```

- Approach adopted in first version of SML.

Overloading Equality

Second Approach

- Make equality fully polymorphic.

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

- Type of member function:

```
member :: [a] -> a -> Bool
```

- Miranda used this approach.
  - Equality applied to a function yields a runtime error.
  - Equality applied to an abstract type compares the underlying representation, which violates abstraction principles.

Overloading Equality

Only provides overloading for ==.

Third Approach

- Make equality polymorphic in a limited way:

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

where $a_{(==)}$ is a type variable ranging *only* over types that admit equality.

- Now we can type the member function:

```
member :: [a(==)] -> a(==) -> Bool

member [2,3] 4 :: Bool
member ['a', 'b', 'c'] 'c' :: Bool
member [\x->x, \x->x + 2] (\y->y *2)  -- type error
```

- Approach used in SML today, where the type $a_{(==)}$ is called an "eqtype variable" and is written ``a.

Type Classes

- Type classes solve these problems. They
  - Allow users to define functions using overloaded operations, eg, `square`, `squares`, and `member`.
  - Generalize ML's eqtypes to arbitrary types.
  - Provide concise types to describe overloaded functions, so no exponential blow-up.
  - Allow users to declare new collections of overloaded functions: equality and arithmetic operators are not privileged.
  - Fit within type inference framework.
  - Implemented as a source-to-source translation.

Intuition

- Sorting functions often take a comparison operator as an argument:

```
qsort:: (a -> a -> Bool) -> [a] -> [a]
qsort cmp [] = []
qsort cmp (x:xs) = qsort cmp (filter (cmp x) xs)
                   ++ [x] ++
                   qsort cmp (filter (not.cmp x) xs)
```

which allows the function to be parametric.

- We can use the same idea with other overloaded operations.

Intuition, continued.

- Consider the "overloaded" function `parabola`:

```
parabola x = (x * x) + x
```

- We can rewrite the function to take the overloaded operators as arguments:

```
parabola' (plus, times) x = plus (times x x) x
```

The extra parameter is a "dictionary" that provides implementations for the overloaded ops.

- We have to rewrite our call sites to pass appropriate implementations for plus and times:

```
y = parabola'(int_plus,int_times) 10
z = parabola'(float_plus, float_times) 3.14
```

Intuition: Better Typing

Type class declarations will generate Dictionary type and accessor functions.

```
-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Accessor functions
get_plus :: MathDict a -> (a->a->a)
get_plus (MkMathDict p t) = p

get_times :: MathDict a -> (a->a->a)
get_times (MkMathDict p t) = t

-- "Dictionary-passing style"
parabola :: MathDict a -> a -> a
parabola dict x = let plus  = get_plus  dict
                      times = get_times dict
                  in plus (times x x) x
```

Intuition: Better Typing

Type class instance declarations generate instances of the Dictionary data type.

```
-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)

-- Dictionary construction
intDict   = MkMathDict intPlus   intTimes
floatDict = MkMathDict floatPlus floatTimes

-- Passing dictionaries
y = parabola intDict   10
z = parabola floatDict 3.14
```

If a function has a qualified type, the compiler will add a dictionary parameter and rewrite the body as necessary.

Type Class Design Overview

- Type class declarations
  - Define a set of operations & give the set a name.
  - The operations `==` and `\=`, each with type `a -> a -> Bool`, form the `Eq a` type class.
- Type class instance declarations
  - Specify the implementations for a particular type.
  - For Int, `==` is defined to be integer equality.
- Qualified types
  - Concisely express the operations required on otherwise polymorphic type.

```
member:: Eq w => w -> [w] -> Bool
```

Qualified Types

"for all types w that support the Eq operations"

```
member:: ∀w. Eq w => w -> [w] -> Bool
```

- If a function works for every type with particular properties, the type of the function says just that:

```
sort      :: Ord a  => [a] -> [a]
serialise :: Show a => a -> String
square    :: Num n  => n -> n
squares   ::(Num t, Num t1, Num t2) =>
                  (t, t1, t2) -> (t, t1, t2)
```

- Otherwise, it must work for any type whatsoever

```
reverse :: [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
```

Type Classes

Works for any type 'n' that supports the Num operations

FORGET all you know about OO classes!

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

The class declaration says what the Num operations are

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

An instance declaration for a type T says how the Num operations are implemented on T's

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

```
plusInt :: Int -> Int -> Int
mulInt  :: Int -> Int -> Int
etc, defined as primitives
```

Compiling Overloaded Functions

When you write this...

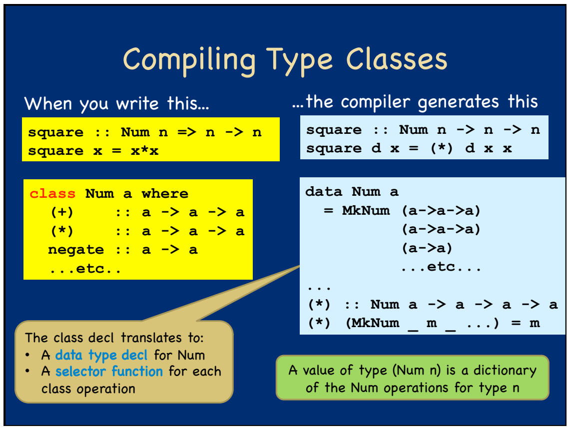
```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

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

The "`Num n =>`" turns into an extra value argument to the function. It is a value of data type `Num n`. This extra argument is a dictionary providing implementations of the required operations.

A value of type (Num n) is a dictionary of the Num operations for type n



Compiling Type Classes

When you write this…

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

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

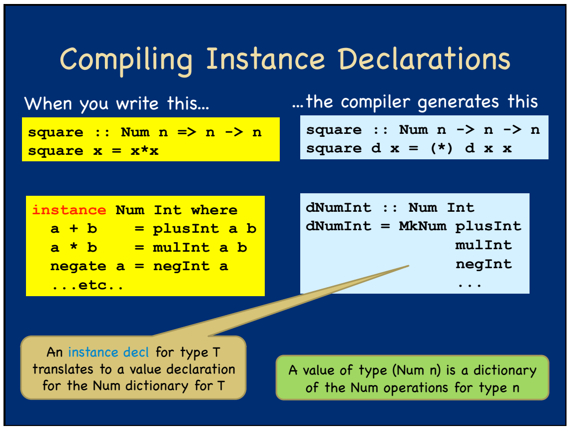
…the compiler generates this

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

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

The class decl translates to:
- A data type decl for Num
- A selector function for each class operation

A value of type (Num n) is a dictionary of the Num operations for type n



Compiling Instance Declarations

When you write this…

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

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

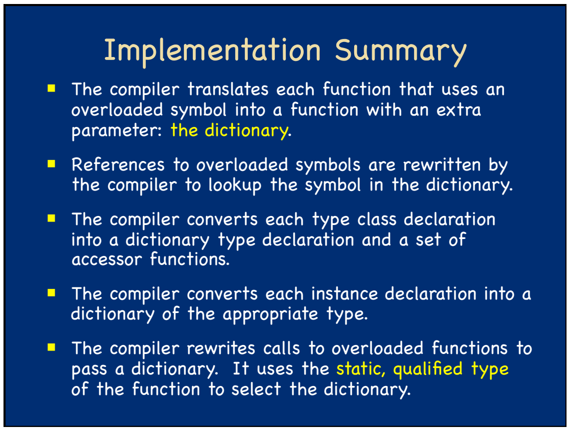
…the compiler generates this

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

```
dNumInt :: Num Int
dNumInt = MkNum plusInt
                mulInt
                negInt
                ...
```

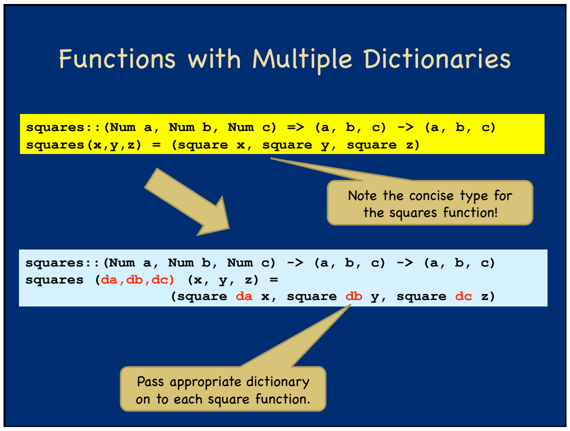
An instance decl for type T translates to a value declaration for the Num dictionary for T

A value of type (Num n) is a dictionary of the Num operations for type n



Implementation Summary

- The compiler translates each function that uses an overloaded symbol into a function with an extra parameter: the dictionary.
- References to overloaded symbols are rewritten by the compiler to lookup the symbol in the dictionary.
- The compiler converts each type class declaration into a dictionary type declaration and a set of accessor functions.
- The compiler converts each instance declaration into a dictionary of the appropriate type.
- The compiler rewrites calls to overloaded functions to pass a dictionary. It uses the static, qualified type of the function to select the dictionary.



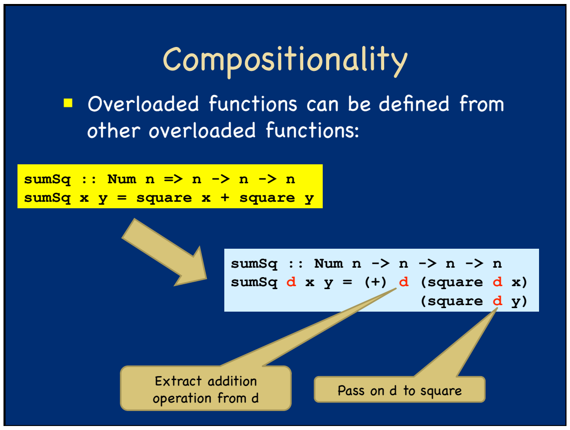
Functions with Multiple Dictionaries

```
squares::(Num a, Num b, Num c) => (a, b, c) -> (a, b, c)
squares(x,y,z) = (square x, square y, square z)
```

Note the concise type for the squares function!

```
squares::(Num a, Num b, Num c) -> (a, b, c) -> (a, b, c)
squares (da,db,dc) (x, y, z) =
                 (square da x, square db y, square dc z)
```

Pass appropriate dictionary on to each square function.



Compositionality

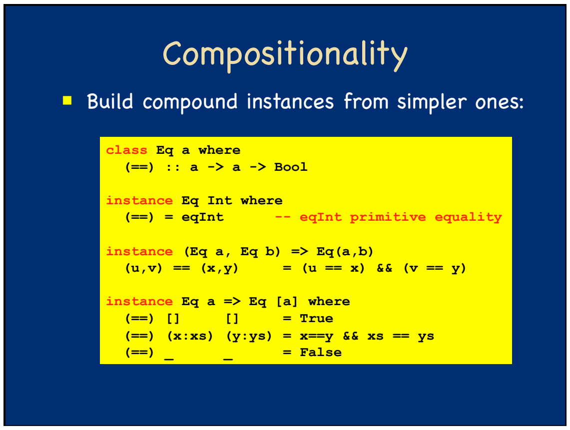
- Overloaded functions can be defined from other overloaded functions:

```
sumSq :: Num n => n -> n -> n
sumSq x y = square x + square y
```

```
sumSq :: Num n -> n -> n -> n
sumSq d x y = (+) d (square d x)
                    (square d y)
```

Extract addition operation from d

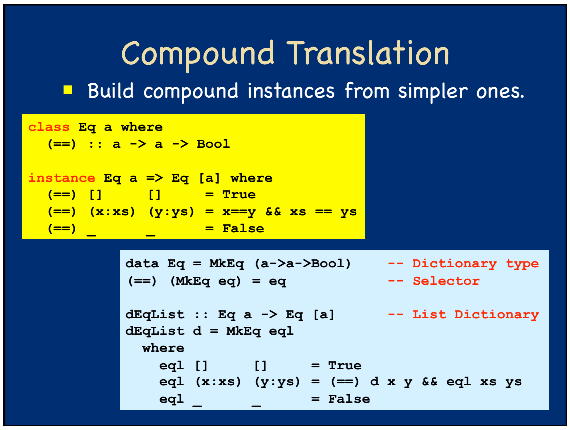
Pass on d to square



Compositionality

- Build compound instances from simpler ones:

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

instance Eq Int where
  (==) = eqInt        -- eqInt primitive equality

instance (Eq a, Eq b) => Eq(a,b)
  (u,v) == (x,y)      = (u == x) && (v == y)

instance Eq a => Eq [a] where
  (==) []     []      = True
  (==) (x:xs) (y:ys)  = x==y && xs == ys
  (==) _      _       = False
```

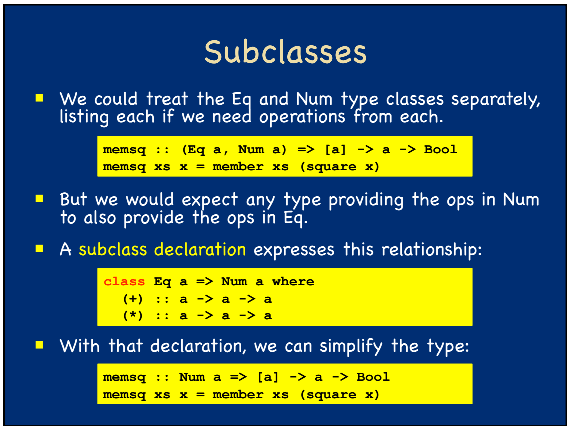


Compound Translation

- Build compound instances from simpler ones.

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

instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _      = False
```

```
data Eq = MkEq (a->a->Bool)     -- Dictionary type
(==) (MkEq eq) = eq             -- Selector

dEqList :: Eq a -> Eq [a]       -- List Dictionary
dEqList d = MkEq eql
  where
    eql []     []     = True
    eql (x:xs) (y:ys) = (==) d x y && eql xs ys
    eql _      _      = False
```



Subclasses

- We could treat the Eq and Num type classes separately, listing each if we need operations from each.

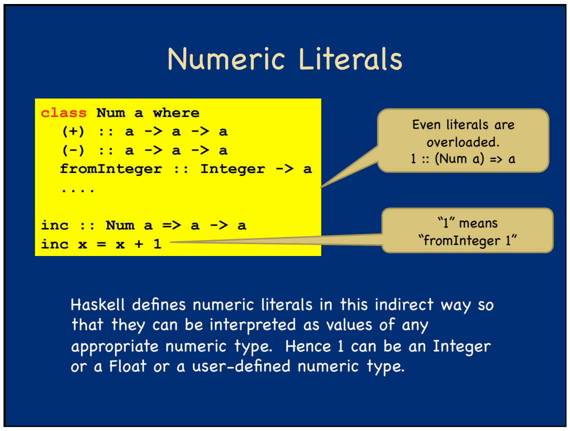
```
memsq :: (Eq a, Num a) => [a] -> a -> Bool
memsq xs x = member xs (square x)
```

- But we would expect any type providing the ops in Num to also provide the ops in Eq.
- A subclass declaration expresses this relationship:

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

- With that declaration, we can simplify the type:

```
memsq :: Num a => [a] -> a -> Bool
memsq xs x = member xs (square x)
```



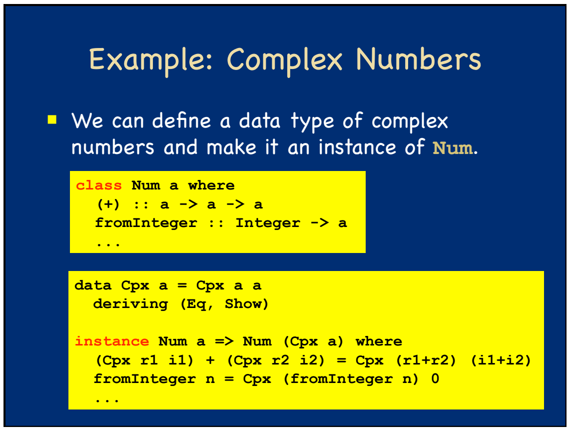
Numeric Literals

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

inc :: Num a => a -> a
inc x = x + 1
```

Even literals are overloaded.
1 :: (Num a) => a

"1" means
"fromInteger 1"

Haskell defines numeric literals in this indirect way so that they can be interpreted as values of any appropriate numeric type. Hence 1 can be an Integer or a Float or a user-defined numeric type.

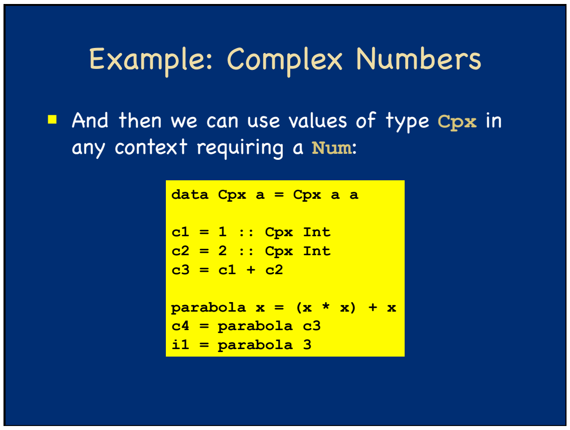


Example: Complex Numbers

- We can define a data type of complex numbers and make it an instance of `Num`.

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

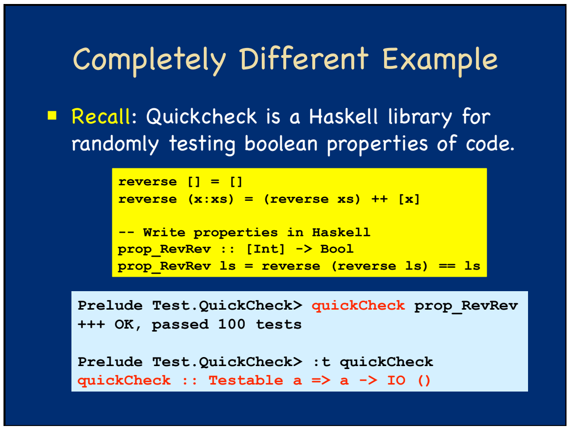
```
data Cpx a = Cpx a a
  deriving (Eq, Show)

instance Num a => Num (Cpx a) where
  (Cpx r1 i1) + (Cpx r2 i2) = Cpx (r1+r2) (i1+i2)
  fromInteger n = Cpx (fromInteger n) 0
  ...
```



Example: Complex Numbers

- And then we can use values of type `Cpx` in any context requiring a `Num`:

```
data Cpx a = Cpx a a

c1 = 1 :: Cpx Int
c2 = 2 :: Cpx Int
c3 = c1 + c2

parabola x = (x * x) + x
c4 = parabola c3
i1 = parabola 3
```



Completely Different Example

- Recall: Quickcheck is a Haskell library for randomly testing boolean properties of code.

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]

-- Write properties in Haskell
prop_RevRev :: [Int] -> Bool
prop_RevRev ls = reverse (reverse ls) == ls
```

```
Prelude Test.QuickCheck> quickCheck prop_RevRev
+++ OK, passed 100 tests

Prelude Test.QuickCheck> :t quickCheck
quickCheck :: Testable a => a -> IO ()
```

Quickcheck Uses Type Classes

```
quickCheck :: Testable a => a -> IO ()

class Testable a where
  test :: a -> RandSupply -> Bool

instance Testable Bool where
  test b r = b

class Arbitrary a where
  arby :: RandSupply -> a

instance (Arbitrary a, Testable b)
           => Testable (a->b) where
  test f r = test (f (arby r1)) r2
           where (r1,r2) = split r
```

```
split :: RandSupply -> (RandSupply, RandSupply)
```

A completely different example: Quickcheck

```
prop_RevRev :: [Int]-> Bool
```

```
class Testable a where
  test :: a -> RandSupply -> Bool

instance Testable Bool where
  test b r = b

instance (Arbitrary a, Testable b)
      => Testable (a->b) where
  test f r = test (f (arby r1)) r2
           where (r1,r2) = split r
```

```
test prop_RevRev r

= test (prop_RevRev (arby r1)) r2
  where (r1,r2) = split r

= prop_RevRev (arby r1)
```

Using instance for (->)

Using instance for Bool

A completely different example: Quickcheck

```
class Arbitrary a where
  arby :: RandSupply -> a

instance Arbitrary Int where
  arby r = randInt r

instance Arbitrary a
      => Arbitrary [a] where
  arby r | even r1 = []
         | otherwise = arby r2 : arby r3
    where
      (r1,r') = split r
      (r2,r3) = split r'
```

Generate Nil value

Generate cons value

```
split :: RandSupply -> (RandSupply, RandSupply)
randInt :: RandSupply -> Int
```

A completely different example: Quickcheck

- QuickCheck uses type classes to auto-generate
  - random values
  - testing functions

  based on the type of the function under test
- Nothing is built into Haskell;
  QuickCheck is just a library!
- Plenty of wrinkles, especially
  - test data should satisfy preconditions
  - generating test data in sparse domains

QuickCheck: A Lightweight tool for random testing of Haskell Programs

---

Many Type Classes

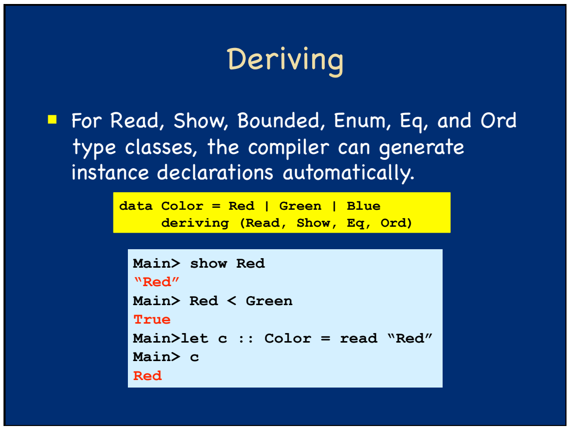
- Eq: equality
- Ord: comparison
- Num: numerical operations
- Show: convert to string
- Read: convert from string
- Testable, Arbitrary: testing.
- Enum: ops on sequentially ordered types
- Bounded: upper and lower values of a type
- Generic programming, reflection, monads, …
- And many more.

Default Methods

- Type classes can define "default methods."

```
class  Eq a  where
   (==), (/=) :: a -> a -> Bool
        -- Minimal complete definition:
        --       (==) or (/=)
   x /= y      =  not (x == y)
   x == y      =  not (x /= y)
```

- Instance declarations can override default by providing a more specific definition.

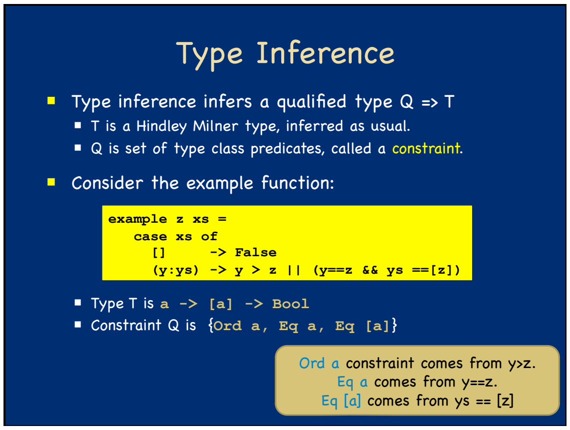


Deriving

- For Read, Show, Bounded, Enum, Eq, and Ord type classes, the compiler can generate instance declarations automatically.

```
data Color = Red | Green | Blue
     deriving (Read, Show, Eq, Ord)
```

```
Main> show Red
"Red"
Main> Red < Green
True
Main>let c :: Color = read "Red"
Main> c
Red
```



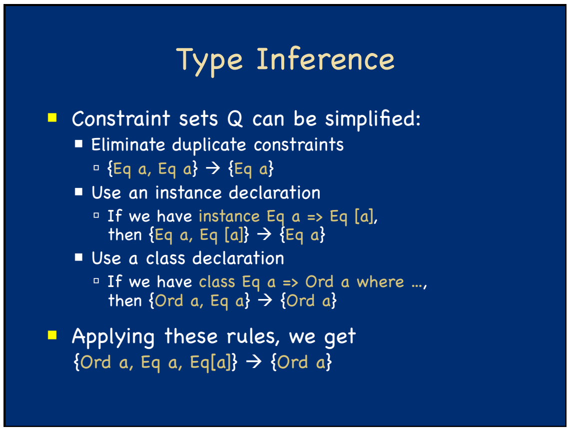
Type Inference

- Type inference infers a qualified type Q => T
  - T is a Hindley Milner type, inferred as usual.
  - Q is set of type class predicates, called a constraint.
- Consider the example function:

```
example z xs =
   case xs of
     []      -> False
     (y:ys)  -> y > z || (y==z && ys ==[z])
```

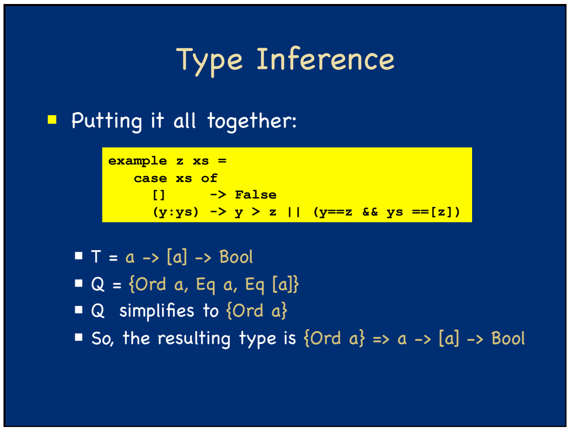
  - Type T is `a -> [a] -> Bool`
  - Constraint Q is {`Ord a, Eq a, Eq [a]`}

Ord a constraint comes from y>z.
Eq a comes from y==z.
Eq [a] comes from ys == [z]



Type Inference

- Constraint sets Q can be simplified:
  - Eliminate duplicate constraints
    - {Eq a, Eq a} → {Eq a}
  - Use an instance declaration
    - If we have instance Eq a => Eq [a], then {Eq a, Eq [a]} → {Eq a}
  - Use a class declaration
    - If we have class Eq a => Ord a where ..., then {Ord a, Eq a} → {Ord a}
- Applying these rules, we get {Ord a, Eq a, Eq[a]} → {Ord a}

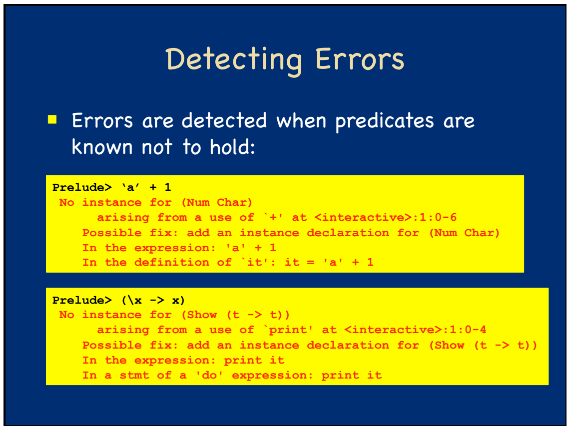


Type Inference

- Putting it all together:

```
example z xs =
   case xs of
     []      -> False
     (y:ys)  -> y > z || (y==z && ys ==[z])
```

  - T = a -> [a] -> Bool
  - Q = {Ord a, Eq a, Eq [a]}
  - Q simplifies to {Ord a}
  - So, the resulting type is {Ord a} => a -> [a] -> Bool

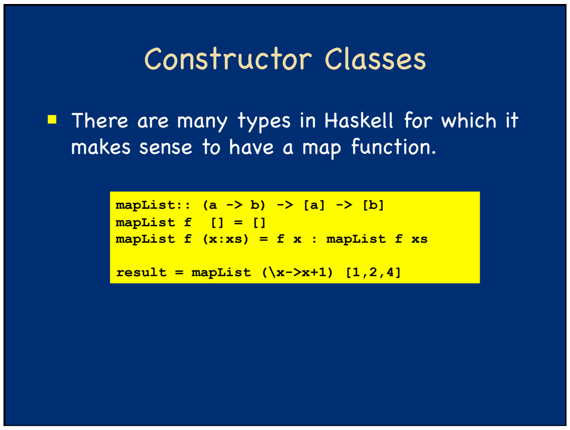


Detecting Errors

- Errors are detected when predicates are known not to hold:

```
Prelude> 'a' + 1
 No instance for (Num Char)
      arising from a use of `+' at <interactive>:1:0-6
    Possible fix: add an instance declaration for (Num Char)
    In the expression: 'a' + 1
    In the definition of `it': it = 'a' + 1
```

```
Prelude> (\x -> x)
 No instance for (Show (t -> t))
      arising from a use of `print' at <interactive>:1:0-4
    Possible fix: add an instance declaration for (Show (t -> t))
    In the expression: print it
    In a stmt of a 'do' expression: print it
```



Constructor Classes

- There are many types in Haskell for which it makes sense to have a map function.

```
mapList:: (a -> b) -> [a] -> [b]
mapList f  [] = []
mapList f (x:xs) = f x : mapList f xs

result = mapList (\x->x+1) [1,2,4]
```

Constructor Classes

- There are many types in Haskell for which it makes sense to have a map function.

```
Data Tree a = Leaf a | Node(Tree a, Tree a)
    deriving Show

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node(l,r)) = Node (mapTree f l, mapTree f r)

t1 = Node(Node(Leaf 3, Leaf 4), Leaf 5)
result = mapTree (\x->x+1) t1
```

Constructor Classes

- There are many types in Haskell for which it makes sense to have a map function.

```
Data Opt a = Some a | None
 deriving Show

mapOpt :: (a -> b) -> Opt a -> Opt b
mapOpt f None = None
mapOpt f (Some x) = Some (f x)

o1 = Some 10
result = mapOpt (\x->x+1) o1
```

Constructor Classes

- All of these map functions share the same structure.

```
mapList :: (a -> b) -> [a] -> [b]
mapTree :: (a -> b) -> Tree a -> Tree b
mapOpt  :: (a -> b) -> Opt a -> Opt b
```

- They can all be written as:

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

- where g is [-] for lists, Tree for trees, and Opt for options.
- Note that g is a function from types to types. It is a *type constructor.*

Constructor Classes

- We can capture this pattern in a *constructor class*, which is a type class where the predicate ranges over type constructors:

```
class HasMap g where
  map :: (a->b) ->(g a -> g b)
```

Constructor Classes

- We can make Lists, Trees, and Opts instances of this class:

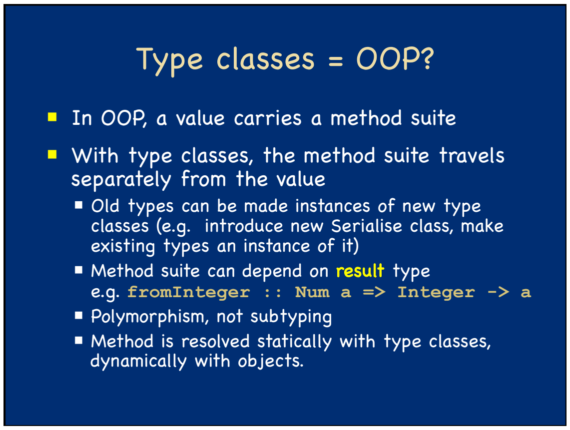
```
class HasMap f where
  map :: (a->b) -> f a -> f b

instance HasMap [] where
  map f [] = []
  map f (x:xs) = f x : map f xs

instance HasMap Tree where
  map f (Leaf x) = Leaf (f x)
  map f (Node(t1,t2)) = Node(map f t1, map f t2)

instance HasMap Opt where
  map f (Some s) = Some (f s)
  map f None = None
```

Constructor Classes

- We can then use the overloaded symbol map to map over all three kinds of data structures:

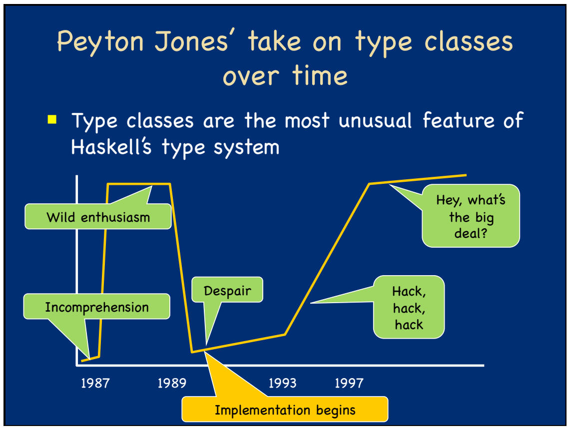
```
*Main> map (\x->x+1) [1,2,3]
[2,3,4]
it :: [Integer]
*Main> map (\x->x+1) (Node(Leaf 1, Leaf 2))
Node (Leaf 2,Leaf 3)
it :: Tree Integer
*Main> map (\x->x+1) (Some 1)
Some 2
it :: Opt Integer
```

- The `HasMap` constructor class is part of the standard Prelude for Haskell, in which it is called "Functor."



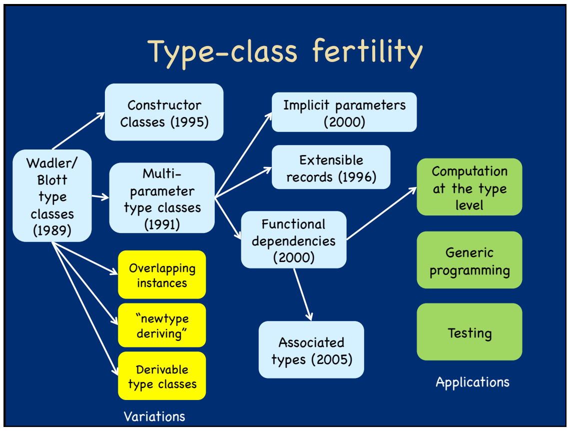
Type classes = OOP?

- In OOP, a value carries a method suite
- With type classes, the method suite travels separately from the value
  - Old types can be made instances of new type classes (e.g. introduce new Serialise class, make existing types an instance of it)
  - Method suite can depend on **result** type e.g. `fromInteger :: Num a => Integer -> a`
  - Polymorphism, not subtyping
  - Method is resolved statically with type classes, dynamically with objects.



Peyton Jones' take on type classes over time

- Type classes are the most unusual feature of Haskell's type system



Type-class fertility

Wadler/ Blott type classes (1989)

Constructor Classes (1995)

Multi-parameter type classes (1991)

Implicit parameters (2000)

Extensible records (1996)

Functional dependencies (2000)

Associated types (2005)

Overlapping instances

"newtype deriving"

Derivable type classes

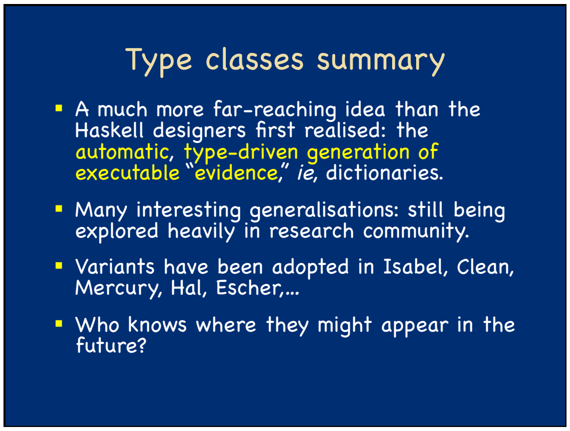
Variations

Computation at the type level

Generic programming

Testing

Applications



Type classes summary

- A much more far-reaching idea than the Haskell designers first realised: the automatic, type-driven generation of executable "evidence," *ie,* dictionaries.
- Many interesting generalisations: still being explored heavily in research community.
- Variants have been adopted in Isabel, Clean, Mercury, Hal, Escher,…
- Who knows where they might appear in the future?