Polymorphism

- Parametric Polymorphism

Example

```
id     ::  a  -> a
length :: [a] -> Int
```

Same definition for all types!

Polymorphism

- Parametric Polymorphism

Example

```
id     ::  a  -> a
length :: [a] -> Int
```

Same definition for all types!

- Ad-hoc Polymorphism

Example

```
(+) :: Int   -> Int   -> Int
(+) :: Float -> Float -> Float
```

Special definition for each type!

Arithmetic

- You can write `4*4` and `0.5*0.5`, but not

  ```
  square x = x*x
  ```

  and then

  ```
  square 4
  square 0.5
  ```

Arithmetic

- You can write `4*4` and `0.5*0.5`, but not

```
square x = x*x
```

  and then

```
square 4
square 0.5
```

- Generate several functions for `square`?

```
square :: Int -> Int
square :: Float -> Float
```

Arithmetic

- You can write `4*4` and `0.5*0.5`, but not

```
square x = x*x
```

and then

```
square 4
square 0.5
```

- Generate several functions for `square`?

```
square :: Int -> Int
square :: Float -> Float
```

- What about

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

Equality

- Same as arithmetic?

  This is ok

  ```
  140 == 15
  'a' == 'b'
  ```

  but this is not

  ```
  member [1,2,3]   2
  member "Haskell" k
  ```

  for

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

Equality

- Fully polymorphic equality?

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

Allows us to write:

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

Equality

- Fully polymorphic equality?

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

Allows us to write:

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

But it fails for lists of functions!

Equality

- Limited polymorphic equality? (SML)

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

`Int` and `Float` are subtypes of `''a`,
but functions are not.

Object-oriented programming

- Each object contains “a dictionary” of methods.

```
square (x,(add,mul,neg)) =
  (mul x x,(add,mul,neg))
```

Object-oriented programming

- Each object contains “a dictionary” of methods.

```
square (x,(add,mul,neg)) =
  (mul x x,(add,mul,neg))


square (2, (addInt, mulInt, negInt))
    == (mulInt 2 2, ...)
square (0.5, (addFloat, mulFloat, negFloat))
    == (mulFloat 0.5 0.5, ...)
```

Object-oriented programming

- Each object contains “a dictionary” of methods.

```
square (x,(add,mul,neg)) =
  (mul x x,(add,mul,neg))
```

```
square (2, (addInt, mulInt, negInt))
    == (mulInt 2 2, ...)
square (0.5, (addFloat, mulFloat, negFloat))
    == (mulFloat 0.5 0.5, ...)
```

```
square :: (a, (a->a->a, a->a->a, a->a))
       -> (a, (a->a->a, a->a->a, a->a))
```

Implementable using parametric polymorphism!

Object-oriented programming (cont.)

```
member []             (y,eq2) = False
member ((x,eq1):xs)   (y,eq2) =
  eq? x y || member xs (y,eq2)
```

Both values carry the equality operator!

- Are they the same?
- Which to choose, if not?

Object-oriented programming (cont.)

```
member []              (y,eq2) = False
member ((x,eq1):xs)  (y,eq2) =
  eq? x y || member xs (y,eq2)
```

Both values carry the equality operator!

- Are they the same?
- Which to choose, if not?

Dictionaries should be passed around independently!?

Example

```
square :: (a->a->a, a->a->a, a->a) -- Num for a
       -> a -> a                   -- operator type
```

Example

```
square :: (a->a->a, a->a->a, a->a)  -- Num for a
       -> a -> a                    -- operator type


member :: (a -> a -> Bool)          -- Eq for a
       -> [a] -> a -> Bool          -- operator type
```

Example

```
square :: (a->a->a, a->a->a, a->a) -- Num for a
       -> a -> a                   -- operator type


member :: (a -> a -> Bool)         -- Eq for a
       -> [a] -> a -> Bool         -- operator type


squares:: (a->a->a, a->a->a, a->a) -- Num for a
       -> (b->b->b, b->b->b, b->b) -- Num for b
       -> (c->c->c, c->c->c, c->c) -- Num for c
       -> (a,b,c) -> (a,b,c)       -- operator type
```

Intuition

- Type classes are predicates over types.
- The predicates ensure that types have certain functionality.
- Class declarations define the kind of functionality.
- Instance declarations define it for the concrete type.

“New” syntax

```
class [context =>] classname tvar where {cbody}
```

- The body of the class decl. contains type signatures and default declarations.

"New" syntax

```
class [context =>] classname tvar where {cbody}
```

- The body of the class decl. contains type signatures and default declarations.

```
instance [context =>] classname type where {ibody}
```

- Type must be in form "tcon tvar_1 … tvar_n", where all type variables must be different.
- Body only contains declarations.

Type classes

Semantics: Using transformation to a form similar to the previous example.

- Describe operator types using a **`class`**.

```
class Num a where
  add, mul :: a -> a -> a
  neg      :: a -> a        -- neg :: Num a => a -> a
```

$\Downarrow$

Type classes

Semantics: Using transformation to a form similar to the previous example.

- Describe operator types using a `class`.

```
class Num a where
  add, mul :: a -> a -> a
  neg      :: a -> a          -- neg :: Num a => a -> a
```

$\Downarrow$

```
data NumD a = NumD (a->a->a) (a->a->a) (a->a)

add (NumD a _ _) = a
mul (NumD _ m _) = m
neg (NumD _ _ n) = n          -- neg :: NumD a -> a -> a
```

Type classes (cont.)

- Describe instances.

```
instance Num Int where
  add = addInt
  mul = mulInt
  neg = negInt
```

$\Downarrow$

Type classes (cont.)

- Describe instances.

```
instance Num Int where
  add = addInt
  mul = mulInt
  neg = negInt
```

$\Downarrow$

```
numDInt :: NumD Int
numDInt =  NumD addInt mulInt negInt
```

Type classes (cont.)

- Describe your own ad-hoc-polymorphic operators.
- Transform the use of operators.

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

$\Downarrow$

Type classes (cont.)

- Describe your own ad-hoc-polymorphic operators.
- Transform the use of operators.

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

⇓

```
square :: NumD a -> a -> a
square numDa x = mul numDa x x
```

Type classes (cont.)

```
mul x y      -- x :: Int
add q w      -- q :: a
```

⇓

Type classes (cont.)

```
mul x y      -- x :: Int
add q w      -- q :: a
```

⇓

```
mul numDInt x y -- x :: Int
add numDa q w   -- q :: a
```

Type classes (cont.)

```
mul x y      -- x :: Int
add q w      -- q :: a
```

⇓

```
mul numDInt x y -- x :: Int
add numDa q w   -- q :: a
```

```
square x = mul x x
```

⇓

Type classes (cont.)

```
mul x y      -- x :: Int
add q w      -- q :: a
```

⇓

```
mul numDInt x y -- x :: Int
add numDa q w   -- q :: a
```

```
square x = mul x x
```

⇓

```
square numDa x = mul numDa x x
```

Polymorphic type classes

```
class Eq a where
  eq :: a -> a -> Bool

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

$\Downarrow$

Polymorphic type classes

```
class Eq a where
  eq :: a -> a -> Bool

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

    ⇓

data EqD a = EqD (a->a->Bool)
eq (EqD e) = e

eqDLa :: EqD a -> EqD [a]
eqDLa eqDa = EqD eq'
  where eq' (x:xs) (y:ys) = eq eqDa x y &&
        eq'                 eq (eqDLa eqDa) xs ys
        eq'     []     [] = True
        eq'     _      _  = False
```

Polymorphic type classes

```
class Eq a where
  eq :: a -> a -> Bool

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

    ⇓

data EqD a = EqD (a->a->Bool)
eq (EqD e) = e
```

Polymorphic type classes

```
class Eq a where
  eq :: a -> a -> Bool

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

    ⇓

data EqD a = EqD (a->a->Bool)
eq (EqD e) = e

eqDLa :: EqD a -> EqD [a]
eqDLa eqDa = EqD eq'
  where eq' (x:xs) (y:ys) = eq eqDa x y &&
        eq'                 eq (eqDLa eqDa) xs ys
        eq'     []     [] = True
        eq'     _      _  = False
```

Subclasses

```
class Eq a where
  eq :: a -> a -> Bool

class Eq a => Num a where
  add, mul :: a -> a -> a
  neg      :: a -> a
```

$\Downarrow$

Subclasses

```
class Eq a where
  eq :: a -> a -> Bool

class Eq a => Num a where
  add, mul :: a -> a -> a
  neg      :: a -> a

    ⇓

data EqD a = EqD (a->a->Bool)
eq (EqD e) = e

data NumD a = NumD (EqD a) (a->a->a) (a->a->a) (a->a)

add (NumD _ a _ _) = a
mul (NumD _ _ m _) = m
neg (NumD _ _ _ n) = n

eqDFromNumD (NumD ed _ _ _) = ed
```

Subclasses (cont.)

```
instance Eq Int where
  eq = eqInt

instance Num Int where
  add = addInt
  mul = mulInt
  neg = negInt
```

$\Downarrow$

Subclasses (cont.)

```
instance Eq Int where
  eq = eqInt

instance Num Int where
  add = addInt
  mul = mulInt
  neg = negInt
```

$\Downarrow$

```
eqDInt :: EqD Int
eqDInt =  EqD eqInt

numDInt :: NumD Int
numDInt =  NumD eqDInt addInt mulInt negInt
```

Default implementations

```
class Eq a where
  eq  :: a -> a -> Bool
  neq :: a -> a -> Bool
  eq  x y = not (neq x y)
  neq x y = not (eq x y)
```

$\Downarrow$

Default implementations

```
class Eq a where
  eq  :: a -> a -> Bool
  neq :: a -> a -> Bool
  eq  x y = not (neq x y)
  neq x y = not (eq x y)
```

$\Downarrow$

```
data EqD a = EqD (a->a->Bool) (a->a->Bool)

eq  (EqD e _) = e
neq (EqD _ n) = n
```

Default implementations (cont.)

```
instance Eq Bool where
  eq = eqBool
```

$\Downarrow$

Default implementations (cont.)

```
instance Eq Bool where
  eq = eqBool

    ⇓

eqDBool = EqD eq′ neq′
  where eq′       = eqBool
        neq′ x y = not (eq eqDBool x y)
```

GHC uses pragmas to specify minimal complete definitions.

Conclusion

Pro:

- Type classes are inferred
- Transforms to parametric polymorphism
- No surprises in binary relations (e.g., equivalence)

Conclusion

Pro:

- Type classes are inferred
- Transforms to parametric polymorphism
- No surprises in binary relations (e.g., equivalence)

Con:

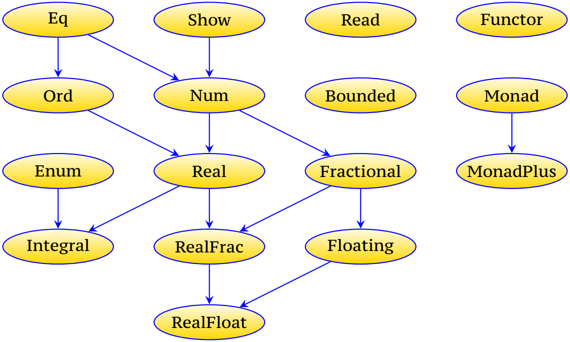
- No general overloading (OOP)
- Forces you to use `newtype`s

Further Reading

- P. Wadler, S. Blott.
  "How to make ad-hoc polymorphism less ad hoc"
  POPL'89

- P. Hudak, J. Peterson, J. Fasel
  "A Gentle Introduction to Haskell"
  `haskell.org/tutorial/stdclasses.html`

Predefined Typeclasses



Predefined Typeclasses

Class Ord

```
class  (Eq a) => Ord a where
    compare                      ::  a -> a -> Ordering
    (<), (<=), (>=), (>)         ::  a -> a -> Bool
    max, min                     ::  a -> a -> a
    compare x y  | x == y        =  EQ
                 | x <= y        =  LT
                 | otherwise     =  GT
    x <= y    = compare x y /= GT
    x <  y    = compare x y == LT
    x >= y    = compare x y /= LT
    x >  y    = compare x y == GT
    max x y   = if x >= y then x else y
    min x y   = if x <  y then x else y
```

Predefined Typeclasses

Class Enum

```
class  Enum a  where
    toEnum           :: Int -> a
    fromEnum         :: a -> Int
    enumFrom         :: a -> [a]                -- [n..]
    enumFromThen     :: a -> a -> [a]           -- [n,n'..]
    enumFromTo       :: a -> a -> [a]           -- [n..m]
    enumFromThenTo   :: a -> a -> a -> [a]      -- [n,n'..m]

succ, pred :: Enum a => a -> a
succ       =  toEnum . (+1) . fromEnum
pred       =  toEnum . (subtract 1) . fromEnum
```

Numbers

Class Num

```
class  (Eq a, Show a) => Num a  where
     (+), (-), (*)      :: a -> a -> a
     negate             :: a -> a
     abs, signum        :: a -> a
     fromInteger        :: Integer -> a

     x - y  =  x + negate y
```

Class Real

```
class  (Num a, Ord a) => Real a  where
    toRational         ::  a -> Rational
```

Numbers

Class Integral

```
class  (Real a, Enum a) => Integral a  where
     quot, rem          :: a -> a -> a
     div, mod           :: a -> a -> a
     quotRem, divMod    :: a -> a -> (a,a)
     toInteger          :: a -> Integer
```

Class Fractional

```
class  (Num a) => Fractional a  where
    (/)                 :: a -> a -> a
    recip               :: a -> a
    fromRational        :: Rational -> a
```

NB!

Additionally we have: RealFrac, Floating, and RealFloat.

Classes Show and Read

Class Show

```
type ShowS = String -> String

class Show a where
    showsPrec  ::  Int -> a -> ShowS
    showList   ::  [a] -> ShowS
    show       ::  a -> String

    show x           =  showsPrec 0 x ""
    showsPrec _ x s  =  show x ++ s
```

Class Read

```
class Read a where ...

read :: (Read a) => String -> a
```

Class Show

Arithmetic

```
data Expr = Num Int        | Var String
          | Expr :+: Expr | Expr :*: Expr

showExp :: Expr -> String
showExp (Num n)        = show n
showExp (Var v)        = v
showExp (e1  :+:  e2) = showExp e1 ++ "+" ++ showExp e2
showExp (e1  :*:  e2) = showArg e1 ++ "*" ++ showArg e2

showArg (e1  :+:  e2) = "(" ++ showExp (e1 :+: e2) ++ ")"
showArg e              = showExp e
```

NB!

Definitions are $O(n^2)$!

Class Show

Predefined helper-functions

```
shows         ::  Show a => a -> ShowS
shows         =   showsPrec 0

showChar      ::  Char -> ShowS
showChar      =   (:)

showString    ::  String -> ShowS
showString    =   (++)

showParen     ::  Bool -> ShowS -> ShowS
showParen b p
   = if b then showChar '(' . p . showChar ')' else p
```

Class Show

Printing of arithmetic expressions

```
instance Show Expr where
  showsPrec _   (Num n)         = shows n
  showsPrec _   (Var v)         = showString v
  showsPrec d   (e1  :+:  e2)   = showParen (d > 0)
                                $ showsPrec 0 e1
                                . showString "+"
                                . showsPrec 0 e2
  showsPrec _   (e1  :*:  e2)   = showsPrec 1 e1
                                . showString "*"
                                . showsPrec 1 e2
```