Session 6

Polymorphism
Overloading
Classes

Universiteit Antwerpen

Polymorphic Types

The function length calculates the length of any list, irrespective of the type of its elements.

```
> length [1,3,5,7]
4

> length ["Yes","No"]
2

> length [isDigit,isLower,isUpper]
3
```

Universiteit Antwerpen

This idea is made precise in the type for length by the inclusion of a type variable:

```
length :: [a] → Int
```

For any type a, length takes a list of values of type a and returns an integer.

A type with variables is called polymorphic.

Universiteit Antwerpen

Note:

- Many of the functions defined in the standard prelude are polymorphic. For example:

```
fst  :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip  :: [a] → [b] → [(a,b)]
```

Universiteit Antwerpen

Overloaded Types

The arithmetic operator + calculates the sum of any two numbers of the same numeric type.

For example:

```
> 1+2
3

> 1.1 + 2.2
3.3
```

Universiteit Antwerpen

This idea is made precise in the type for + by the inclusion of a class constraint:

```
(+) :: Num a ⇒ a → a → a
```

For any type a in the class Num of numeric types, + takes two values of type a and returns another.

A type with constraints is called overloaded.

Intuitive Distinction: Polymorphism<>Overloading

- Polymorphism
  - Client can call function (method) with different actual parameter types
  - In Functional Programming: ANY type
  - One implementation of the function takes different (poly) forms (morph) at call-time
  - Can be perceived as template mechanism
- Overloading
  - Formal parameters are constrained
  - Their type space is partitioned
  - For each partition, there can be a different body

Universiteit Antwerpen

Exercise

Polymorphism

- Define a higher order function “composeFafterG” for composing unary functions

- Apply it to execute “composeFafterG mySqrt mySquare 5” with

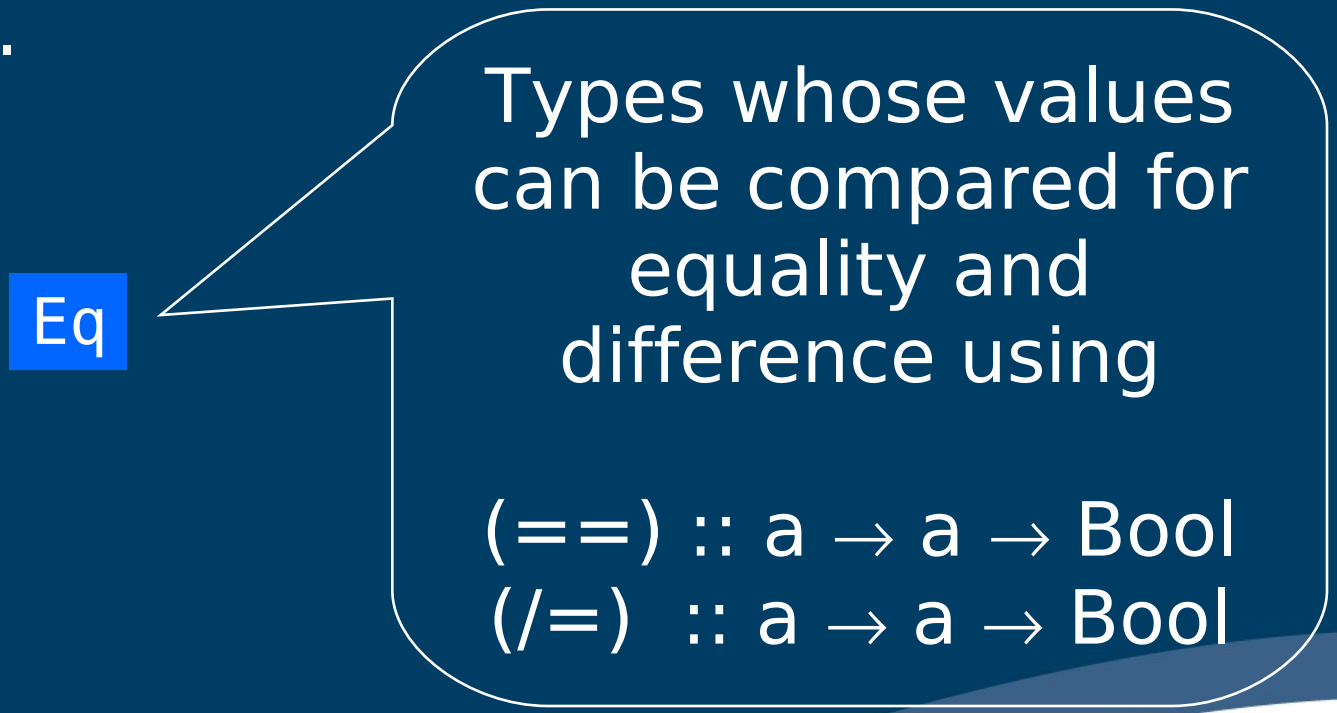
```
mySquare::Integer->Integer
mySquare x=x*x

mySqrt::Integer->Float
mySqrt x = sqrt(fromIntegral x)
```

Universiteit Antwerpen

Classes in Haskell

A class is a collection of types that support certain operations, called the methods of the class.



Haskell has a number of basic classes, including:

- `Eq` - Equality types
- `Ord` - Ordered types
- `Show` - Showable types
- `Read` - Readable types
- `Num` - Numeric types

Example: class Enum (1/2)

```
module Prelude (
  -- Modules
    module PreludeList, module PreludeText, module PreludeIO,
  ...
  -- Types
    Bool(False, True),
    Maybe(Nothing, Just),
  ...
  -- Classes declared
    Enum(succ, pred, toEnum, fromEnum, enumFrom, enumFromThen,
         enumFromTo, enumFromThenTo),
  ...
  -- Classes defined
class  Enum a  where
    succ, pred       :: a -> a
    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]
```

Universiteit Antwerpen

Example: class Enum (2/2)

```
        -- Minimal complete definition:
        --      toEnum, fromEnum
--
-- NOTE: these default methods only make sense for types
--   that map injectively into Int using fromEnum
--  and toEnum.
    succ            =  toEnum . (+1) . fromEnum
    pred            =  toEnum . (subtract 1) . fromEnum
    enumFrom x      =  map toEnum [fromEnum x ..]
    enumFromTo x y  =  map toEnum [fromEnum x .. fromEnum y]
    enumFromThen x y =  map toEnum [fromEnum x, fromEnum y ..]
    enumFromThenTo x y z =
                    map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

think of “fromEnumToInt”, e.g. “fromCharToInt”

“..” syntacic sugar for `enumFromTo` which is hardcoded for `Int`:

```
instance Enum Integer where ...
```

Therefore we delegate to the `Int` domain for all other *Enums*.

implemented in C

Universiteit Antwerpen

Instantiation of a Class

```
instance  Enum Char  where
   toEnum           = primIntToChar
   fromEnum         = primCharToInt
   enumFrom c       = map toEnum [fromEnum c .. fromEnum
     (maxBound::Char)]
   enumFromThen c c' = map toEnum [fromEnum c, fromEnum c' ..
     fromEnum lastChar]
                where lastChar :: Char
                     lastChar | c' < c    = minBound
                              | otherwise = maxBound

instance  Enum Float  where
   succ x         =  x+1
   pred x         =  x-1
   toEnum         =  fromIntegral
   fromEnum       =  fromInteger . truncate   -- may overflow
   enumFrom       =  numericEnumFrom
   enumFromThen   =  numericEnumFromThen
   enumFromTo     =  numericEnumFromTo
   enumFromThenTo   =  numericEnumFromThenTo
```

Sufficient to conform to class `Enum`. `succ` and `prec` are "reused"

Universiteit Antwerpen

Inheritance (Specialization)

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

-- Minimal complete definition: (==) or (/=)

   x /= y     =  not (x == y)
   x == y     =  not (x /= y)
```

Ord subclasses Eq

```
class  (Eq a) => Ord a  where
   compare              :: a -> a -> Ordering
   (<), (<=), (>=), (>) :: a -> a -> Bool
   max, min             :: a -> a -> a

      -- Minimal complete definition:
      --      (<=) or compare
      -- Using compare can be more efficient
         for complex types.
   compare x y
       | x == y    =  EQ
       | x <= y    =  LT
       | otherwise =  GT
```

compare is defined in terms of <=

```
   x <= y           =  compare x y /= GT
   x <  y           =  compare x y == LT
   x >= y           =  compare x y /= LT
   x >  y           =  compare x y == GT

-- note: (min x y, max x y) = (x,y) or (y,x)

   max x y
       | x <= y    =  y
       | otherwise =  x
   min x y
       | x <= y    =  x
       | otherwise =  y
```

```
data  Ordering  =  LT | EQ | GT
    deriving (Eq, Ord, Enum, Read,
    Show, Bounded)
```

Universiteit Antwerpen

Example methods:

```
(==) :: Eq a    ⇒ a → a → Bool

(<)  :: Ord a   ⇒ a → a → Bool

show :: Show a ⇒ a → String

read :: Read a ⇒ String → a

(*)  :: Num a   ⇒ a → a → a
```

Universiteit Antwerpen

Exercises

(1) What are the types of the following functions?

```
second xs        = head (tail xs)
swap (x,y)       = (y,x)
pair x y         = (x,y)
double x         = x*2
palindrome xs    = reverse xs == xs
twice f x        = f (f x)
```

(2) Check your answers using Hugs.

Universiteit Antwerpen

Exercises (cont.)

Define a type Colour with possible values Green, AlmostGreen and Blue.

(3) Using as little code as possible, add (==) and (\=) operators which consider all colours to be different

(4) Using as little code as possible, add (==) and (\=) operators which consider “Green” and “AlmostGreen” to be the same.

Universiteit Antwerpen