Functional Programming

2022-2023

Sjaak Smetsers

Type classes revisited
Lecture 6

Outline

- Type classes
- Overloading vs. higher-order functions
- Polymorhic type inference

Overloading

- sometimes we wish to use the same name for semantically different, but related functions
  - +, * etc: arithmetic operations (Int, Integer, Float, Double . . .)
  - (==), (/=) : equality and inequality (almost any type)
  - show, read: converting to and from strings (almost any type)
- we want to overload these identifiers
- Haskell's type classes: a systematic approach to overloading
  - (ad-hoc polymorphism vs universal polymorphism)

Class declarations

- new classes can be declared using the `class` mechanism.
- eg the class `Eq` of equality types is declared in the standard prelude as follows:

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

- this declaration states that for a type `a` to be an instance of the class `Eq`, it must support equality and inequality operators of the specified types.
- `(==),(/=)` are member functions of the type class Eq (also called methods)
- types of the member functions:

```
(==),(/=):: (Eq a) ⇒ a → a → Bool
```

- `(Eq a)` ⇒ is a class context; it constrains the type variable `a`

Overloaded functions

- since == is overloaded, `x == y` can be ambiguous (i.e we don't know which instance is used here)
- what happens if the compiler can't resolve overloading?
- eg list membership uses equality:

```
elem :: (Eq a) ⇒ a → [a] → Bool
elem x [ ]        = False
elem x (y : ys) = x == y || elem x ys
```

- elem becomes overloaded
- in general: a (polymorphic) function is called *overloaded* if its type contains one or more class contexts (aka *class constraints*)

Default definitions

- inequality is typically defined in terms of equality (or vice versa)

```
class Eq a where
  (==),(/=):: a ⟶ a ⟶ Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

- *default declarations* avoid having to give both definitions every time we introduce a new instance
  - in an instance declaration of `Eq` it suffices now to provide *either* the code for `==` *or* the code for `/=`

Subclasses

- classes can be extended

```
data Ordering = LT | EQ | GT
class (Eq a) ⇒ Ord a where
    compare              :: a ⟶ a ⟶ Ordering
    (<), (<=), (>), (>=) :: a ⟶ a⟶ Bool
    max, min             :: a ⟶ a ⟶ a
```

- Ord is a subclass of Eq; conversely, Eq is a superclass of Ord
- subclasses keep class contexts manageable
- necessary if method of superclass is used in one of the default methods
  - eg the default implementation of compare is

```
compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT
```

- Ord includes several default implementations
  - defining either compare or ≤ is sufficient

Bounded

- instances of `Ord` have to implement a *total* order
- occasionally, a type has a *least* and a *greatest* element with respect to that ordering

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

- the type `Int` of machine integers is bounded, the type `Integer` of mathematical integers isn't

```
>>> maxBound :: Int
9223372036854775807
>>> maxBound :: Integer
No instance for Bounded Integer
```

- (it's a *compile-time* error to use `maxBound` at `Integer`)

Enum

- the dot-dot notation is overloaded

```
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]
```

- useful for generating test data

```
⋙ [Mon .. Sun]
[Mon, Tue, Wed, Thu, Fri, Sat, Sun]
```

Instance declarations

- the type

```
data Blood = A | B | AB | O
```

- can be made into an equality type as follows:

```
instance Eq Blood where
  A  == A  = True
  B  == B  = True
  AB == AB = True
  O  == O  = True
  _  == _  = False
```

Class instances of parametric types

- to define equality on a parametric type, say, `Tree a` we require equality on the element type `a`
- an instance declaration can have a context too

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
instance (Eq a) ⇒ Eq (Tree a) where
  Leaf x1     == Leaf x2     = x1 == x2
  Leaf _      == Fork _ _    = False
  Fork _ _    == Leaf _      = False
  Fork l1 r1  == Fork l2 r2  = l1 == l2 && r1 == r2
```

- read: if `a` supports equality, then `Tree a` supports equality too

Deriving instances

- defining equality (or instances of some other classes) is tedious, can be derived automatically:

```
data Gender = Female | Male
  deriving (Eq, Ord, Enum, Show, Read)
```

- the compiler generates the ‘obvious’ code (using a technique similar to generic programming; lecture 7)
- deriving works for parametric types too

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
  deriving (Eq, Ord, Show, Read)
```

Pretty printing

- converting data into textual representation: pretty printing

```
type ShowS = String ⟶ String

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

  show x     = showsPrec 0 ""
```

- operator precedences can be taken into account
- for each type we can also decide how to format lists of elements of that type
- you almost always want to say **`deriving (Show)`**

Parsing

- converting textual representation into data

```
type ReadS a = String → [(a,String)]
class Read a where
  readsPrec :: Int → ReadS a
  readList  :: ReadS [a]
```

- Read uses "list of successes" technique (more in lecture 13: Parsing)
- Additionally we have

```
read :: Read a ⇒ String → a
```

- `read:` input string must be completely consumed
- `read.show` should be the identity

Overloading vs. hio-functions (I)

- instead of overloading we can use functions as arguments
- eg

```
elem :: (Eq a) ⇒ a ⟶ [a] ⟶ Bool
elem x [ ]      = False
elem x (y : ys) = x == y || elem x ys
```

- abstract away from Eq

```
elemBy :: (a ⟶ a ⟶ Bool) ⟶ a ⟶ [a] ⟶ Bool
elemBy eq x [ ]      = False
elemBy eq x (y : ys) = x `eq` y || elemBy eq x ys
```

Overloading vs. hio-functions (II)

- instance of Eq for [ ]:

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

- eliminating/abstracting away from Eq

```
eqList :: (a ⟶ a ⟶ Bool) ⟶ [a] ⟶ [a] ⟶ Bool
eqList eq []     []     = True
eqList eq []     _l     = False
eqList eq _l     []     = False
eqList eq (x:xs) (y:ys) = x `eq` y && eqList eq xs ys
```

Overloading vs. hio-functions (III)

- consider type

```
data Gtree a = Branch a [Gtree a]
```

- instance of Eq:

```
instance (Eq a) ⇒ Eq (Gtree a) where
    Branch e1 trs1 == Branch e2 trs2 = e1 == e2 && trs1 == trs2
```

- eliminating overloading

```
eqGtree :: (a → a → Bool) → Gtree a → Gtree a → Bool
eqGtree eq (Branch e1 trs1) (Branch e2 trs2)
    = e1 `eq` e2 && eqList (eqGtree eq) trs1 trs2
```

Overloading in Haskell's standard libraries

- For many overloaded functions there exists a higher-order variant

```
sort :: Ord a => [a] -> [a]
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
maximum :: Ord a => [a] -> a
maximumBy :: (a -> a -> Ordering) -> [a] -> a
group :: Eq a => [a] -> [[a]]
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
```

- Some useful utility functions

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
comparing :: Ord a => (b -> a) -> b -> b -> Ordering
```

Example: sortBy

```
data Person = Person { name::String, age::Integer, course::String }
  deriving (Show)
```

- sort by name

```
sortByName = sortBy (\p1 p2 -> name p1 `compare` name p2)
```

- sort by name using `comparing`

```
sortByName = sortBy (comparing name)
```

- sort by decreasing age

```
sortByDecrAge = sortBy (\p1 p2 -> age p2 `compare` age p1)
```

- sort by decreasing age using `on`

```
sortByDecrAge = sortBy (flip compare `on` age)
```

nub **more efficient** (I)

- the nub function eliminates duplicate values from a list.

```
nub :: (Eq a) => [a] -> [a]
```

- eg.

```
>>> nub [1,5,3,9,3,9,7,10,1,6,5]
[1,5,3,9,7,10,6]
```

- the time complexity is O(N^2)
- can improve efficiency if the elements are ordered

```
nubEffi :: Ord a => [a] -> [a]
nubEffi = map head . group . sort
```

- time complexity is O(N log N)
- however

```
>>> nubEffi [1,5,3,9,3,9,7,10,1,6,5]
[1,3,5,6,7,9,10]
```

nub **more efficient (II)**

- using the Set library

```
nubSet :: Ord a => [a] -> [a]
nubSet = Set.toList . Set.fromList
```

- keep the original order

```
nubKeep = map snd . sortBy (comparing fst) . map head .
            groupBy ((==) `on` snd) . sortBy (comparing snd) .
            zip [1..]
```

- much, much better: use `nubOrd` from `Data.List.Extra`

```
nubOrd :: Ord a => [a] -> [a]
```

Classes or Algebraic Data Types (ADTs)? (I)

- Modeling with ADTs
- An animal can be either a dog or a cat. We can model this with an ADT

```
type Name   = String
data Animal = Dog Name | Cat Name

makeSound :: Animal -> [Char]
makeSound (Dog name) = name ++ " says: woof, woof"
makeSound (Cat name) = name ++ " says: meow, meow"
```

- eg

```
>>> makeSound (Dog "Baxter")
"Baxter says: woof, woof"
```

Classes or Algebraic Data Types (ADTs)? (II)

- Modeling with classes

```haskell
type Name = String

data Dog = Dog Name
data Cat = Cat Name

class Animal a where
  makeSound :: a -> String

instance Animal Dog where
  makeSound (Dog name) = name ++ " says: woof, woof"

instance Animal Cat where
  makeSound (Cat name) = name ++ " says: meow, meow"
```

- eg

```
>>> makeSound (Cat "Milo")
"Milo says: meow, meow"
```

Classes or Algebraic Data Types (ADTs)? (III)

- What is the difference?
  - The ADT-based solution is *closed:* the set of cases is fixed (eg. defined in one place).
  - The class-based solution is *open:* we can add easily new cases without changing anything else (eg. even in other modules).
- In the class-based solution, for example, another module could introduce a pig:

```
data Pig = Pig Int
instance Animal Pig where
  makeSound (Pig _weight) = "Piggy says: oink, oink"
```

- A closed abstraction is better when we want to handle multiple cases in a single function:

```
getAlong :: Animal -> Animal -> Bool
getAlong (Cat _) (Pig _) = True
getAlong (Dog _) (Pig _) = True
getAlong (Dog _) (Dog _) = True
getAlong (Pig _) _       = True
getAlong _       _       = False
```

Polymorhic type inference

- How do you solve the type inference puzzle puzzle?
- Before typing a function `f`, examine all functions used by `f` first.
  - start with a general type for each function
  - use patterns, guards and right-hand sides to derive more specific type information
  - introduce a fresh type for each polymorphic function (a new placeholder for each type variable)
- Type inference yields the most general type (MGT) of a function: Every valid type signature for a function is an instance of its MGT.

Polymorphic type inference: `twice`

```
twice :: …
twice f x = f (f x)
```

Polymorphic type inference: `twice`

```
twice :: ❶ ⟶ ❷ ⟶ ❸
twice f x = f (f x)
```

```
f   :: ❶ = ?
x   :: ❷ = ?
rhs :: ❸ = ?
```

Polymorphic type inference: `twice`

```
twice :: (4 ⟶ 5) ⟶ 4 ⟶ 3
twice f x = f @ (f @ x)
```

- Making the invisible application operator visible

```
@ :: (a⟶b) ⟶ a ⟶ b
```

```
f   :: 1 = ?
x   :: 2 = ?
rhs :: 3 = ?
@   :: (4 ⟶ 5)⟶ 4 ⟶ 5
1 = 4 ⟶ 5
2 = 4
```

Polymorphic type inference: twice

```
twice :: (6 ⟶ 7) ⟶ 6 ⟶ 7
twice f x = f @ (f @ x)
```

```
@ :: (a⟶b) ⟶ a ⟶ b
```

```
f   :: 1 = ?
x   :: 2 = ?
rhs :: 3 = 7
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
1 = 6 ⟶ 7
2 = 6
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
4 ⟶ 5 = 6 ⟶ 7
5 = 7
4 = 6
```

Polymorphic type inference: `twice`

```
twice :: (7 ⟶ 7) ⟶ 7 ⟶ 7
twice f x = f @ (f @ x)
```

```
@ :: (a⟶b) ⟶ a ⟶ b
```

```
f   :: 1 = ?
x   :: 2 = ?
rhs :: 3 = 7
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
1 = 6 ⟶ 7
2 = 6
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
4 ⟶ 5 = 6 ⟶ 7
5 = 7
4 = 6
6 = 7
```

Polymorphic type inference: `twice`

```
twice :: (7 ⟶ 7) ⟶ 7 ⟶ 7
twice f x = f @ (f @ x)
```

$$@ :: (a \longrightarrow b) \longrightarrow a \longrightarrow b$$

- No further restrictions: 7 remains to be 'unknown'

```
f   :: 1 = ?
x   :: 2 = ?
rhs :: 3 = 7
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
1 = 6 ⟶ 7
2 = 6
@   :: (6 ⟶ 7)⟶ 6 ⟶ 7
4 ⟶ 5 = 6 ⟶ 7
5 = 7
4 = 6
6 = 7
```

Polymorphic type inference: `twice`

```
twice :: (a ⟶ a) ⟶ a ⟶ a
twice f x = f (f x)
```

Polymorphic type inference: f6 (previous exam)

```
f6 :: ?
f6 xs = reverse [(y,x) | (x,y) ← xs]
```

Polymorphic type inference: f6 (previous exam)

```
f6 :: ❶ ⟶ ❷
f6 xs = reverse [(y,x) | (x,y) ⟵ xs]
```

```
reverse :: [a] ⟶ [a]
```

```
xs     :: ❶ = ?
rhs    :: ❷ = ?
left ⟵  :: ❸
right ⟵ :: [❸]
❶ = [❸]
```

Polymorphic type inference: `f6` (previous exam)

```
reverse :: [a] ⟶ [a]
```

```
f6 :: [❸] ⟶ ❷
f6 xs = reverse [(y,x) | (x,y) ⟵ xs]
```

```
xs       :: ❶ = ?
rhs      :: ❷ = ?
left ⟵  :: ❸
right ⟵ :: [❸]
❶ = [❸]
❸ = (❹,❺)
x     :: ❹ = ?
y     :: ❺ = ?
```

Polymorphic type inference: f6 (previous exam)

```
f6 :: [(4,5)] → [6]
f6 xs = reverse [(y,x) | (x,y) ← xs]
```

```
reverse :: [a] → [a]
```

```
xs      :: 1 = ?
rhs     :: 2 = [6]
left ←  :: 3
right ← :: [3]
1 = [3]
3 = (4,5)
x     :: 4 = ?
y     :: 5 = ?
reverse :: [6]→[6]
```

Polymorphic type inference: f6 (previous exam)

```
reverse :: [a] ⟶ [a]
```

```
f6 :: [(❹,❺)] ⟶ [(❺, ❹)]
f6 xs = reverse [(y,x) | (x,y) ⟵ xs]
```

```
xs    :: ❶ = ?
rhs   :: ❷ = [❻]
left ⟵ :: ❸
right ⟵ :: [❸]
❶ = [❸]
❸ = (❹,❺)
x    :: ❹ = ?
y    :: ❺ = ?
reverse :: [❻]⟶[❻]
❻ = (❺, ❹)
```

Polymorphic type inference: f6 (previous exam)

```
f6 :: [(a,b)] → [(b,a)]
f6 xs = reverse [(y,x) | (x,y) ← xs]
```

Polymorphic type inference: an overloaded function

```
map :: (a ⟶ b) ⟶ [a] ⟶ [b]
(.) :: (b ⟶ c) ⟶ (a ⟶ b) ⟶ (a ⟶ c)
(>) :: (Ord a) ⟹ a ⟶ a ⟶ Bool
```

```
g :: …
g = map . (>)
```

Polymorphic type inference: an overloaded function

```
g :: ❶
g = map . (>)
```

```
map :: (a → b) → [a] → [b]
(.) :: (b → c) → (a → b) → (a → c)
(>) :: (Ord a) ⇒ a → a → Bool
```

```
(.) :: (❸ → ❹) → (❷ → ❸) → (❷ → ❹)
map :: (❺ → ❻) → [❺] → [❻]
(>) :: (Ord ❼) ⇒ ❼ → ❼ → Bool
```

Polymorphic type inference: an overloaded function

```
g :: ❶
g = map . (>)
```

```
map :: (a ⟶ b) ⟶ [a] ⟶ [b]
(.) :: (b ⟶ c) ⟶ (a ⟶ b) ⟶ (a ⟶ c)
(>) :: (Ord a) ⟹ a ⟶ a ⟶ Bool
```

```
(.) :: (❸ ⟶ ❹) ⟶ (❷ ⟶ ❸) ⟶ (❷ ⟶ ❹)
map :: (❺ ⟶ ❻) ⟶ [❺] ⟶ [❻]
(>) :: (Ord ❼) ⟹ ❼ ⟶ ❼ ⟶ Bool
left arg (.): ❸ = ❺ ⟶ ❻
              ❹ = [❺] ⟶ [❻]
```

Polymorphic type inference: an overloaded function

```
g :: ❶
g = map . (>)
```

```
map :: (a ⟶ b) ⟶ [a] ⟶ [b]
(.) :: (b ⟶ c) ⟶ (a ⟶ b) ⟶ (a ⟶ c)
(>) :: (Ord a) ⟹ a ⟶ a ⟶ Bool
```

```
(.) :: (❸ ⟶ ❹) ⟶ (❷ ⟶ ❸) ⟶ (❷ ⟶ ❹)
map :: :: (❺ ⟶ ❻) ⟶ [❺] ⟶ [❻]
(>) :: (Ord ❼) ⟹ ❼ ⟶ (❼ ⟶ Bool)
left arg (.): ❸ = ❺ ⟶ ❻
              ❹ = [❺] ⟶ [❻]
right arg (.): ❷ = ❼
               ❸ = ❼ ⟶ Bool
```

Polymorphic type inference: an overloaded function

```
map :: (a ⟶ b) ⟶ [a] ⟶ [b]
(.) :: (b ⟶ c) ⟶ (a ⟶ b) ⟶ (a ⟶ c)
(>) :: (Ord a) ⟹ a ⟶ a ⟶ Bool
```

```
g :: ①
g = map . (>)
```

```
(.) :: (③ ⟶ ④) ⟶ (② ⟶ ③) ⟶ (② ⟶ ④)
map :: (⑤ ⟶ ⑥) ⟶ [⑤] ⟶ [⑥]
(>) :: (Ord ⑦) ⟹ ⑦ ⟶ ⑦ ⟶ Bool
left arg (.): ③ = ⑤ ⟶ ⑥
              ④ = [⑤] ⟶ [⑥]
right arg (.): ② = ⑦
               ③ = ⑦ ⟶ Bool
result(.): ① = ② ⟶ ④ = ⑦ ⟶ [⑤] ⟶ [⑥]
```

Polymorphic type inference: an overloaded function

```
g :: (Ord ⑦) ⟹ ⑦ ⟶ [⑤] ⟶ [⑥]
g = map . (>)
```

```
map :: (a ⟶ b) ⟶ [a] ⟶ [b]
(.) :: (b ⟶ c) ⟶ (a ⟶ b) ⟶ (a ⟶ c)
(>) :: (Ord a) ⟹ a ⟶ a ⟶ Bool
```

```
(.) :: (③ ⟶ ④) ⟶ (② ⟶ ③) ⟶ (② ⟶ ④)
map :: (⑤ ⟶ ⑥) ⟶ [⑤] ⟶ [⑥]
(>) :: (Ord ⑦) ⟹ ⑦ ⟶ (⑦ ⟶ Bool)
left arg (.): ③ = ⑤ ⟶ ⑥
              ④ = [⑤] ⟶ [⑥]
right arg (.): ② = ⑦
               ③ = ⑦ ⟶ Bool
result(.): ① = ② ⟶ ④ = ⑦ ⟶ [⑤] ⟶ [⑥]
⑤ = ⑦
⑥ = Bool
```

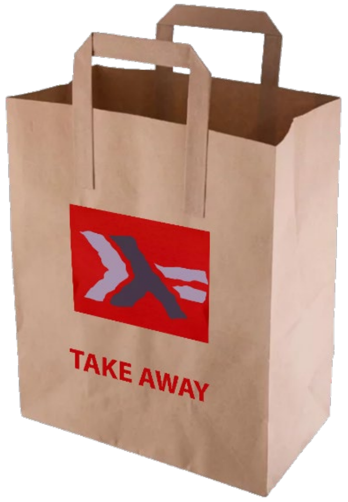
Polymorphic type inference: an overloaded function

```
g :: (Ord a) ⇒ a ⟶ [a] ⟶ [Bool]
g = map . (>)
```

```
map :: (a ⟶ b) ⟶ [a] ⟶ [b]
(.) :: (b ⟶ c) ⟶ (a ⟶ b) ⟶ (a ⟶ c)
(>) :: (Ord a) ⇒ a ⟶ a ⟶ Bool
```

```
(.) :: (❸ ⟶ ❹) ⟶ (❷ ⟶ ❸) ⟶ (❷ ⟶ ❹)
map :: (❺ ⟶ ❻) ⟶ [❺] ⟶ [❻]
(>) :: (Ord ❼) ⇒ ❼ ⟶ (❼ ⟶ Bool)
left arg (.): ❸ = ❺ ⟶ ❻
              ❹ = [❺] ⟶ [❻]
right arg (.): ❷ = ❼
               ❸ = ❼ ⟶ Bool
result(.): ❶ = ❷ ⟶ ❹ = ❼ ⟶ [❺] ⟶ [❻]
❺ = ❼
❻ = Bool
```

Abstraction, abstraction, abstraction



- question: what are the three most important concepts in programming?
- answer: abstraction, abstraction, abstraction!
- type classes allow you to capture commonalities across datatypes
- classes are most useful if the type uniquely determines the instance
- higher-order functions give you greater flexibility