Functional Programming
Type Classes — Overloading in Haskell

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

Overloading

Remember previous classes?

We were able to use

- equality == and ordering < with many different types
- arithmetic operations with many different types

Overloading

Remember previous classes?

We were able to use

- equality == and ordering < with many different types
- arithmetic operations with many different types

Overloading

The **same operator** can be used to execute **different code** at **many different types**.

Overloading

Remember previous classes?

We were able to use

- equality == and ordering < with many different types
- arithmetic operations with many different types

Overloading

The **same operator** can be used to execute **different code** at **many different types**.

Contrast with

Overloading

Remember previous classes?

We were able to use

- equality == and ordering < with many different types
- arithmetic operations with many different types

Overloading

The **same operator** can be used to execute **different code** at **many different types**.

Contrast with

Parametric polymorphism

The **same code** can execute at **many different types**.

Haskell integrates overloading with polymorphism

Restricted polymorphism

- Some functions work on parametric types, but are restricted to specific instances
- Types contain type variables and **constraints**

Haskell integrates overloading with polymorphism

Restricted polymorphism

- Some functions work on parametric types, but are restricted to specific instances
- Types contain type variables and **constraints**

Examples

```
1  -- elem x xs : is x an element of list xs?
2  -- type a must have equality
3  elem :: Eq a => a -> [a] -> Bool
4  -- insert x xs : insert x into sorted list xs
5  -- type a must have comparison
6  insert :: Ord a => a -> [a] -> [a]
7  -- square x : compute the square of x
8  -- type a has numeric operations
9  square :: Num a => a -> a
```

Type classes

- Each constraint mentions a **type class**
  like Eq, Ord, Num, . . .
- A type class specifies a set of operations for a type
  e.g. Eq requires == and /=
- Type classes form a hierarchy
  e.g. Ord a => Eq a
- Many classes are predefined, but you can roll your own

Classes and Instances

- A class declaration **only** specifies a signature (i.e., the class members and their types)

```
1 class Num a where
2   (+), (∗), (−) :: a −> a −> a
3   negate, abs, signum :: a −> a
4   fromInteger :: Integer −> a
```

- An instance declaration specifies that a type belongs to a class by giving definitions for all class members

```
1 instance Num Int where ...
2 instance Num Integer where ...
3 instance Num Double where ...
4 instance Num Float where ...
```

- This info can be obtained from GHCI by

```
1 :i Num
```

Equality

The type class Eq

```
1  class Eq a where
2    (==), (/=) :: a -> a -> Bool
3    x /= y = not (x == y) -- default definition
```

An instance must only provide (==).

Equality

The type class Eq

```
1  class Eq a where
2    (==), (/=) :: a -> a -> Bool
3    x /= y = not (x == y) -- default definition
```

An instance must only provide (==).

Instances of Eq

```
1  instance Eq Int -- Defined in 'GHC.Classes'
2  instance Eq Float -- Defined in 'GHC.Classes'
3  instance Eq Double -- Defined in 'GHC.Classes'
4  instance Eq Char -- Defined in 'GHC.Classes'
5  instance Eq Bool -- Defined in 'GHC.Classes'
6  {- and many more -}
```

Equality

The type class Eq

```
1  class Eq a where
2    (==), (/=) :: a -> a -> Bool
3    x /= y = not (x == y) -- default definition
```

An instance must only provide (==).

Instances of Eq

```
1  instance Eq Int -- Defined in 'GHC.Classes'
2  instance Eq Float -- Defined in 'GHC.Classes'
3  instance Eq Double -- Defined in 'GHC.Classes'
4  instance Eq Char -- Defined in 'GHC.Classes'
5  instance Eq Bool -- Defined in 'GHC.Classes'
6  {- and many more -}
```

Question

Does equality make sense at every type?

Defining Eq for pairs

When are two pairs equal?

Defining Eq for pairs

When are two pairs equal?

Solution

```
1 instance (Eq a, Eq b) => Eq (a, b) where
2    (a1, b1) == (a2, b2) = a1 == a2 && b1 == b2
```

Defining Eq for pairs

When are two pairs equal?

Solution

```
1 instance (Eq a, Eq b) => Eq (a, b) where
2   (a1, b1) == (a2, b2) = a1 == a2 && b1 == b2
```

Is this definition recursive?

Defining Eq for pairs

When are two pairs equal?

Solution

```
1  instance (Eq a, Eq b) => Eq (a, b) where
2    (a1, b1) == (a2, b2) = a1 == a2 && b1 == b2
```

Is this definition recursive?

NO!

Defining Eq for lists

When are two lists equal?

Defining Eq for lists

When are two lists equal?

Solution

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

Defining Eq for lists

When are two lists equal?

Solution

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

Is this definition recursive?

Defining Eq for lists

When are two lists equal?

Solution

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

Is this definition recursive?

YES!
The equality xs == ys.

Handwriting vs deriving an instance

Remember the Hearts game

```
1  data Color = Black | Red
2     deriving (Show)
```

Handwriting vs deriving an instance

Remember the Hearts game

```
1 data Color = Black | Red
2    deriving (Show)
```

Define your own equality

```
1 instance Eq Color where
2    Black == Black = True
3    Red == Red = True
4    _ == _ = False
```

Handwriting vs deriving an instance

Remember the Hearts game

```
data Color = Black | Red
   deriving (Show)
```

Define your own equality

```
instance Eq Color where
   Black == Black = True
   Red == Red = True
   _ == _ = False
```

Same result as deriving Eq

```
data Color = Black | Red
   deriving (Show, Eq)
```

Further useful classes

Show and Read

```
1  class Show a where
2    show :: a −> String
3    {− ... −}
4
5  class Read a where
6    read :: String −> a
7    {− ... −}
```

- Predefined for most built-in types
- Derivable for most datatype definitions

The Ord class (derivable)

```
1  class Eq a => Ord a where
2    compare :: a -> a -> Ordering
3    (<) :: a -> a -> Bool
4    (<=) :: a -> a -> Bool
5    (>) :: a -> a -> Bool
6    (>=) :: a -> a -> Bool
7    max :: a -> a -> a
8    min :: a -> a -> a
9
10 data Ordering = LT | EQ | GT  -- Defined in 'GHC.Types'
```

More classes for you to investigate

- `Enum` (derivable)
- `Bounded` (derivable)

Ambiguity

Some combinations of overloaded functions can lead to ambiguity

```
1  f x = read (show x)
2  g x = show (read x)
```

Ambiguity

Some combinations of overloaded functions can lead to ambiguity

```
1 f x = read (show x)
2 g x = show (read x)
```

What are types of `f` and `g`?

Ambiguity

Some combinations of overloaded functions can lead to ambiguity

```
1  f x = read (show x)
2  g x = show (read x)
```

What are types of f and g?

Solution

```
1  f :: (Read a, Show b) => b -> a
2  g :: String -> String
```

Further pitfalls / features

- Definitions without arguments and without type signatures are not overloaded (monomorphism restriction)
- Numeric literals are overloaded at type `Num a => a`
- Haskell has a **defaulting** mechanism that resolves violations of the monomorphism restriction
- Caveat: GHCi behaves differently than code in a file

Wrapup

Type classes

- provides a signature for an abstract data type
- instances provide implementations at unrelated types
- many classes are predefined and derivable
- pervasively used in Haskell / some pitfalls