Functional Programming

Type Classes — Overloading in Haskell

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2024/25

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

Constrained polymorphism

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

Haskell integrates overloading with polymorphism

Constrained polymorphism

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

Examples

```haskell
-- elem x xs : is x an element of list xs?
-- type a must have equality
elem :: Eq a => a -> [a] -> Bool
-- insert x xs : insert x into sorted list xs
-- type a must have comparison
insert :: Ord a => a -> [a] -> [a]
-- square x : compute the square of x
-- type a has numeric operations
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* specifies a signature (i.e., the class members and their types)

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

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

```haskell
instance Num Int where ...
instance Num Integer where ...
instance Num Double where ...
instance Num Float where ...
```

- This info can be obtained from GHCI by

```haskell
:i Num
```

Equality

The type class Eq

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

An instance must only provide (==).

Equality

The type class Eq

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

An instance must only provide `(==)`.

Instances of Eq

```haskell
instance Eq Int -- Defined in 'GHC.Classes'
instance Eq Float -- Defined in 'GHC.Classes'
instance Eq Double -- Defined in 'GHC.Classes'
instance Eq Char -- Defined in 'GHC.Classes'
instance Eq Bool -- Defined in 'GHC.Classes'
{- and many more -}
```

Equality

The type class Eq

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

An instance must only provide `(==)`.

Instances of Eq

```haskell
instance Eq Int -- Defined in 'GHC.Classes'
instance Eq Float -- Defined in 'GHC.Classes'
instance Eq Double -- Defined in 'GHC.Classes'
instance Eq Char -- Defined in 'GHC.Classes'
instance Eq Bool -- Defined in 'GHC.Classes'
{- and many more -}
```

Tacit assumption

Equality is a congruence relation.

Defining `Eq` for pairs

When are two pairs equal?

Defining Eq for pairs

When are two pairs equal?

Solution

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

Defining Eq for pairs

When are two pairs equal?

Solution

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

Is this definition recursive?

Defining Eq for pairs

When are two pairs equal?

Solution

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

Is this definition recursive?

YES: on types, NO: on values

Defining `Eq` for lists

When are two lists equal?

Defining Eq for lists

When are two lists equal?

Solution

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

Defining Eq for lists

When are two lists equal?

Solution

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

Is this definition recursive?

Defining Eq for lists

When are two lists equal?

Solution

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

Is this definition recursive?

YES: no types, YES: on values

The equality `xs == ys`.

Handwriting vs deriving an instance

Remember the Hearts game

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

Handwriting vs deriving an instance

Remember the Hearts game

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

Define your own equality

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

```
class Show a where
  show :: a -> String
  {- ... -}

class Read a where
  read :: String -> a
  {- ... -}
```

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

The Ord class (derivable)

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

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

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

Ambiguity

Some combinations of overloaded functions can lead to ambiguity

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

What are types of `f` and g?

Ambiguity

Some combinations of overloaded functions can lead to ambiguity

```haskell
f x = read (show x)
g x = show (read x)
```

What are types of `f` and g?

Solution

```haskell
f :: (Read a, Show b) => b -> a
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

- provide 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