Fachhochschule
Nordwestschweiz

Functional Programming

Type Classes

Daniel Kröni
University of Applied Sciences Northwestern Switzerland

Learning Targets

You understand the concept of overloading

You can declare type classes and instances

You know the basic classes provided by the standard library

Content

- Motivation
- Class Declaration
- Instance Declaration
- Standard Classes: Eq, Ord, Show, Num, ...
- Deriving Type Classes

If Haskell would not have overloading

- **Membership check in a list of Bool:**

```
containsBool :: Bool -> [Bool] -> Bool
containsBool _ []      = False
containsBool x (y:ys) = sameBool x y || containsBool x ys
```

  – Type definition

```
data Bool = False | True
```

  – sameBool is the equality function over Bool

```
sameBool :: Bool -> Bool -> Bool
sameBool True  True  = True
sameBool False False = True
sameBool _     _     = False
```

If Haskell would not have overloading

- **Membership check in a list of Colors:**

```
containsColor :: Color -> [Color] -> Bool
containsColor _ []     = False
containsColor x (y:ys) = sameColor x y || containsColor x ys
```

  – Type definition

```
data Color = Red | Green | Blue
```

  – sameColor is the equality function over Color

```
sameColor :: Color -> Color-> Bool
sameColor Red   Red   = True
sameColor Green Green = True
sameColor Blue  Blue  = True
sameColor _     _     = False
```

If Haskell would not have overloading

- **Solution attempt:**
  - Make the equality function a parameter of a general function

```
containsGen :: (a -> a -> Bool) -> a -> [a] -> Bool
containsGen _ _ []      = False
containsGen f x (y:ys) = f x y || containsGen f x ys
```

- **Problems**
  - Too general, any function (a -> a -> Bool) could be passed
  - Each time containsGen is used the equality function needs to be passed explicitly which is making programs less easy to read

```
containsGen sameBool True [False, False, False, True]
```

If Haskell would not have overloading

- **Preferably we would write this signature**

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

  - But this signature is too general!
  - Not for every type equality is defined (e.g. Integer -> Integer)
  - The type variable **a** needs to be restricted to those types which provide an equality operator

- **We need a means to express that a type provides certain functions / operators i.e. a type implements a desired interface**

Type Classes and Instances

- **Class definition**

```haskell
class Compare a where
  same :: a -> a -> Bool
```

  – A type class defines an interface or signature which has to be implemented for a type to belong to the class.

- **Instance definition**

```haskell
instance Compare Bool where
  same True True   = True
  same False False = True
  same _     _     = False
```

  – A type is made a member of a type class by providing an instance definition for that type class.

Class Assertions

- **Type variables can be restricted to only be instantiable with types which are members of a required class**

```
contains :: Compare a => a -> [a] -> Bool
```

Class Constraint

  - Only types which provide an instance for the class Compare can be used
  - Bool for example is accepted, because Bool is an instance of Compare

```
contains True [False, False] ~> False
```

  - ([Int] -> Int) is not accepted: functions can not be compared in general

```
*Main> contains length [sum, product]
<interactive>:16:1:
    No instance for (Compare ([Int] -> Int))
      arising from a use of ‘contains’
    In the expression: contains length [sum, product]
    In an equation for ‘it’: it = contains length [sum, product]
```

Type Classes Applied

- **Having type classes, contains can be defined like this:**

```
contains :: Compare a => a -> [a] -> Bool
contains _ []      = False
contains x (y:ys) = same x y || contains x ys
```

  - Compare a ensures, that the corresponding same operation is available
  - Overloading: We use the same name for a function (same) but its behavior is different depending on the particular type (sameBool, sameColor)

- **Advantages**
  - Reuse: The definition of contains can be used over all types with equality
  - Readability: It's much easier to to read same than sameBool
    This argument holds particular for arithmetic operators $2 +_{Int} 3 *_{Int} 5$

Class Constraints and Context

- **A context can consist of multiple class constraints**

```
showIfSame :: (Eq a, Show a) => a -> a -> String
showIfSame a1 a2 | a1 == a2  = show a1
                 | otherwise = "Not the same"
```

  – a1 == a2 requires a to be in the class Eq
  – show a1 requires a to be in the class Show

- **A context can be used in instance definitions**

```
instance (Eq a) => Eq [a] where
  as == bs = (length as == length bs)
             && (and (zipWith (==) as bs))
```

  – Lists with elements of type a can be compared for equality only if a can be compared for equality

Haskell Type Classes vs. Java Interfaces

- **Class definition**

```
class Hashable a where
  compHash :: a -> Int
```

- **Type definition**

```
data P = P Int
```

- **Instance definition**

```
instance Hashable P where
  compHash (P i) = i
```

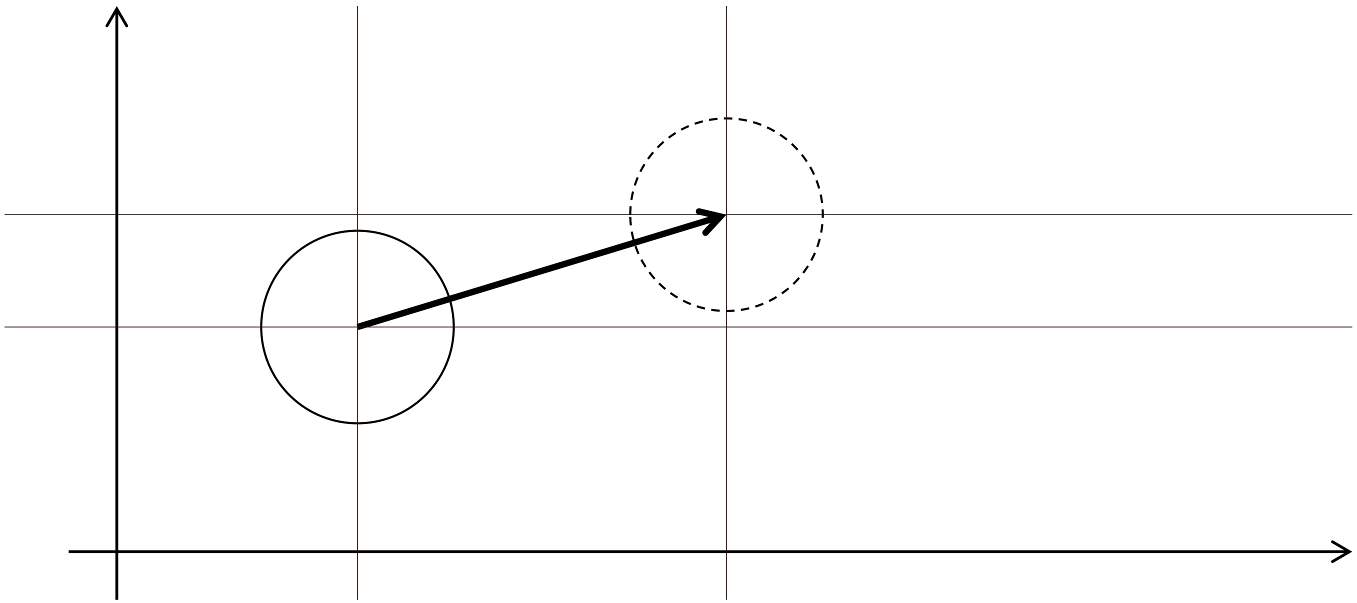
- **Interface definition**

```
interface Hashable {
  int compHash();
}
```

- **Class definition**

```
class P implements Hashable {
  int i = 12;
  int compHash() {
     return i;
  }
}
```

Worksheet: Movable Figures Pt. 1



Basic classes

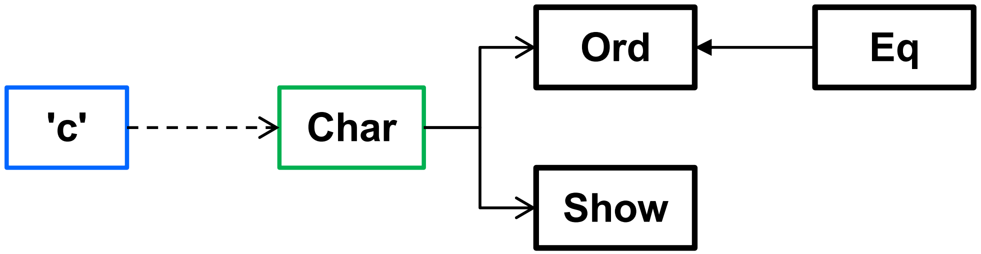
- **Eq – equality types**
  - Contains types whose values can be compared for equality and inequality
  - methods: (==), (/=)
- **Ord – ordered types**
  - Contains types whose values are totally ordered
  - methods: (<), (<=), (>), (>=), min, max
- **Show – showable types**
  - Contains types whose values can be converted into strings of characters
  - method show :: a -> String

Basic classes

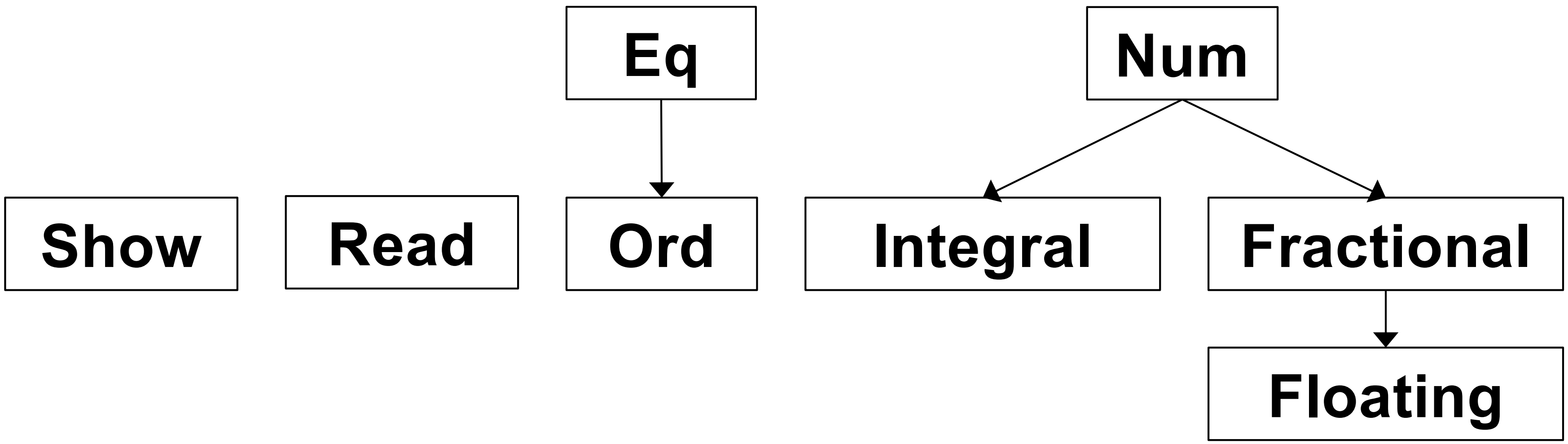
- **Num –** numeric types
  - Contains types whose values are numeric
  - methods: (+), (-), (*), negate, abs, signum
- **Integral –** integral types
  - Contains types that are numeric but of integral value
  - methods: div, mod
- **Fractional –** fractional types
  - Contains types that are numeric but of fractional value
  - methods: (/), recip

Classes, Types and Values

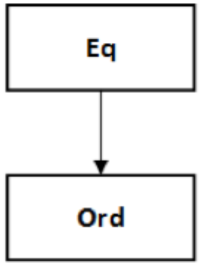
- A value (e.g. 'c') has exactly one type (Char).
- A type can be a member of a **class** (Ord, Show).
- A **class** can be a subclass of another **class** (Ord <: Eq)
  types of Class Ord are also types of class Eq. All methods that a type in Eq has are also available for a type in Ord.



Relations between basic classes



Relations between Eq and Ord




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

Default implementations:
Implement one of both

Ord is a subclass of Eq:
To be an Ord member a type
has to provide an Eq instance

Uses Eq's methods

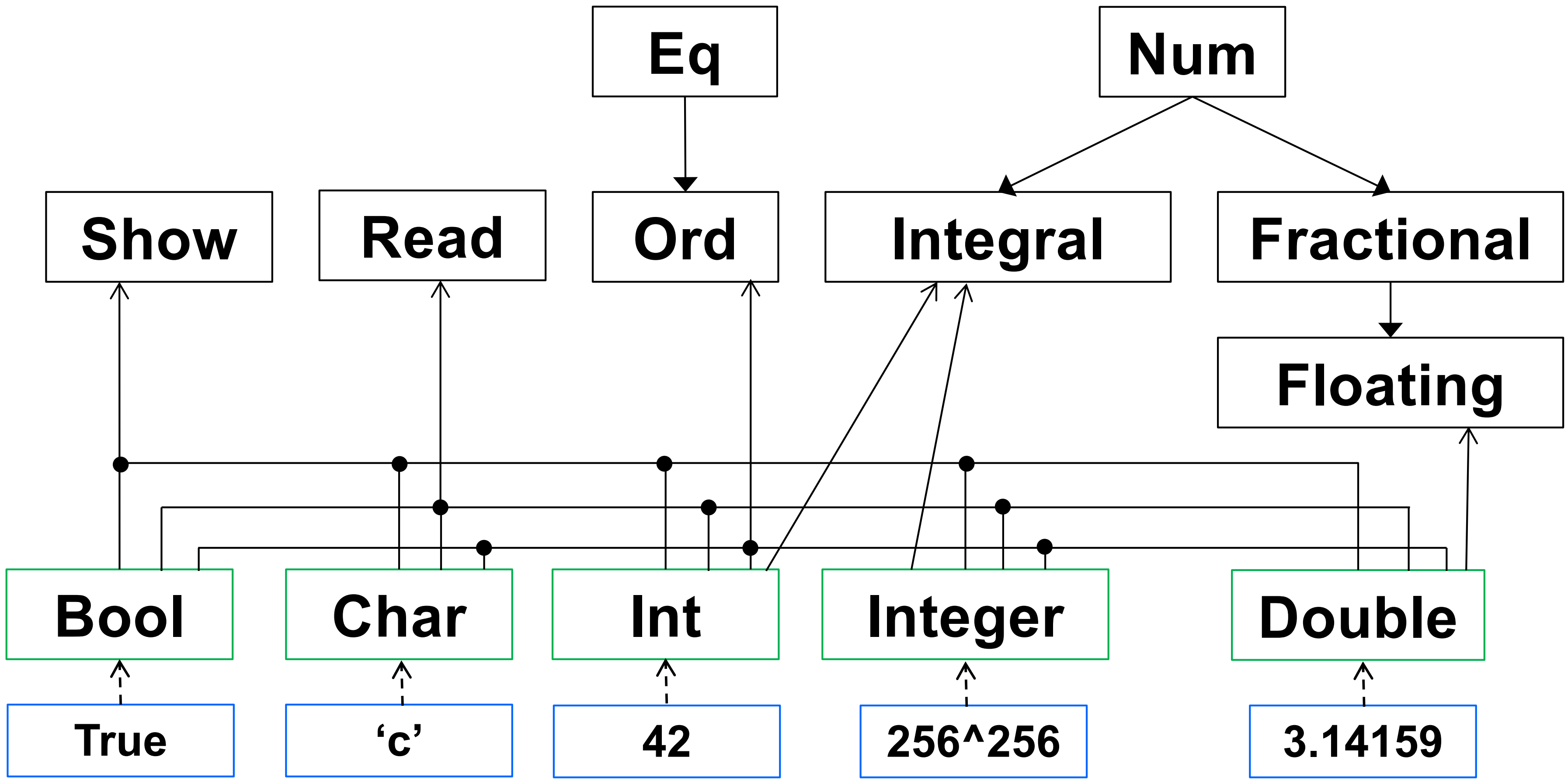
Only <= needs to
be implemented

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

    x <  y = x <= y && x /= y
    x >  y = not (x <= y)
    x >= y = not (x <= y) || x == y
```

Simplified code

All basic types are of class Ord, Show and Read



Deriving Type Classes

- **Sometimes an instance definition is obvious:**

```
data Color = Red | Green | Blue
instance Show Color where
  show Red   = "Red"
  show Green = "Green"
  show Blue  = "Blue"
```

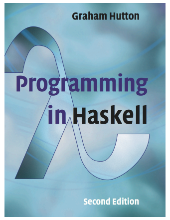
- **Some instances can be automatically derived**

```
data Color = Red | Green | Blue deriving (Show)
```

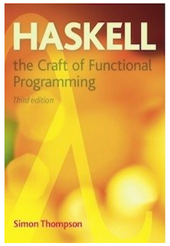
  - Show instance is automatically generated
  - This is supported for Eq, Ord, Show and others

Worksheet: Movable Figures Pt. 2

Further Reading



Chapter 8.5



Chapter 14

Chapter 7

http://learnyouahaskell.com/types-and-typeclasses