universität innsbruck

WS 2021/2022

Functional Programming

Week 6 – Type Classes

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitinger Christian Sternagel

Department of Computer Science

Last Lecture

- layout rule: define blocks via indentation or via `{ ...; ...; ...}`
- case-expressions: perform pattern matching in right-hand sides of defining equations

  ```
  case expr of { pat -> expr; ... ; pat -> expr }
  ```
- local definitions with `let` and `where`

  ```
  let { pat = expr; fName pat ... pat = expr } in expr

  fName pat ... pat = expr
    where  pat                  = expr
           fName pat .. pat = expr
  ```
- guarded equations

  ```
  fName pat ... pat
    | cond = expr
    | ...  = ...    -- + optional where-block
  ```
- recursion on numbers

Type Classes – Definition

Type Classes so Far

- brief introduction that there are type classes, e.g., `Num a`, `Eq a`, ...
- type classes are used to provide uniform access to functions that can be implemented differently for each type
- example
  - `(<) :: Ord a => a -> a -> Bool` is name of function for comparing two elements
  - each of the following types have a different implementation of `(<)`
    - `(<) :: Int -> Int -> Bool`
    - `(<) :: Char -> Char -> Bool`
    - `(<) :: Bool -> Bool -> Bool`
    - `(<) :: Ord a => [a] -> [a] -> Bool`
    - `(<) :: (Ord a, Ord b) => (a, b) -> (a, b) -> Bool`
- upcoming: definition of type classes
  - understand definition of existing type classes
  - specify new type classes
- upcoming: instantiating type classes
  - define an implementation for some type and some type class

Type Classes – Definition

- type classes are defined via the keyword `class`:

  ```
  class TCName a where
    fName :: ty     -- type ty + description of fName
    ...
    lhs = rhs       -- optional default implementation
    ...
  ```

  where
  - `TCName` is a name for the type class, starting with uppercase letter
  - `a` is a single type variable
  - there are (several) type definitions for functions – without defining equations!
  - for each function `fName` there should be some informal description
  - there can be default implementations for each specified function `fName`
- when adding a type constraint `TCName a => ...`, then all functions `fName` are available
- defining a type class instance for some type requires implementation of all functions
- exception: functions that have default implementation can, but do not have to be implemented

Type Classes – Example `Equality`

```
class Equality a where
  equal :: a -> a -> Bool       -- equality
  different :: a -> a -> Bool   -- inequality
  -- properties:
  --   equal x x should evaluate to True
  --   equal and different should be symmetric
  --   exactly one of equal x y and different x y should be True
  equal x y = not (different x y)    -- default implementation
  different x y = not (equal x y)    -- default implementation
```

- if type constraint `Equality b` is added to type of function `f`, then both `equal :: b -> b -> Bool` and `different :: b -> b -> Bool` can be used in defining equation of `f`
- if concrete type `Ty` is an instance of `Equality`, then both `equal :: Ty -> Ty -> Bool` and `different :: Ty -> Ty -> Bool` can be used without adding type constraint
- in order to make some type an instance of `Equality`, at least one of `equal`, `different` has to be implemented for that type

Operator Syntax, Type Class Eq

- Eq is already predefined type class for equality
- only difference to Equality: operators are used instead of function names
- in Haskell every operator can be turned into a function and vice versa
  - parentheses turn arbitrary operator & into function name (&)
  - `a & b` is the same as `(&) a b`
  - `(&) :: ty` is used to specify the type of an operator
  - backticks turn some arbitrary function name fName into an operator `` `fName` ``
  - `fName a b` is the same as `` a `fName` b ``
- consequence: in the following definition (==) and (/=) are just function names

```
class Eq a where
  (==) :: a -> a -> Bool  -- equality
  (/=) :: a -> a -> Bool  -- inequality
  x == y = not (x /= y)   -- default implementation
  x /= y = not (x == y)   -- default implementation
```

http://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html#t:Eq

Type Class Hierarchies

- type classes can be defined hierarchically via type constraints
- syntax: `class (TClass1 a, ..., TClassN a) => TClassNew a where ...`
- consequences
  - type constraint `TClassNew a` implicitly adds type constraints `(TClass1 a, ..., TClassN a)`
  - when adding type constraint `TClassNew a`, all functions that are defined in one of `TClassNew`, `TClass1`, ..., `TClassN` become available
  - an instantiation of `TClassNew` for some type is only possible if that type is already an instance of all of `TClass1`, ..., `TClassN`
  - default implementations in `TClassNew` can make use of functions of `TClass1`, ..., `TClassN`

Example: Type Class Ord

```haskell
class Eq a => Ord a where
  compare :: a -> a -> Ordering -- data Ordering = LT | EQ | GT
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  x < y = x <= y && x /= y
  x > y = y < x
  ...
```

- minimal complete definition: `compare` or `(<=)`
- note: default definition refers to `Eq` function `(/=)`
- http://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html#t:Ord

Type Class Instances

- many types are instances of `Eq` and `Ord`
- examples
  - `Eq Int` meaning: `Int` is an instance of `Eq`
  - `Eq Char`, `Eq Integer`, `Eq Bool`, ...
  - `Eq a => Eq [a]`
    - meaning: lists of `a` are an instance of `Eq` whenever `a` is an instance of `Eq`
    - implication `Eq String`, `Eq [Int]`, `Eq [[Integer]]`, ...
  - `Eq a => Eq (Maybe a)`, `(Eq a, Eq b) => Eq (Either a b)`, ...
  - `Eq ()`, `(Eq a, Eq b) => Eq (a,b)`, ... for tuples of at most 15 entries
  - `Ord Bool`, `Ord Char`, `Ord Integer`, `Ord Double`, ...
  - `Ord a => Ord [a]`, `(Ord a, Ord b) => Ord (Either a b)`, ...
  - `Ord ()`, `(Ord a, Ord b) => Ord (a,b)`, ... for tuples of at most 15 entries
  - `Ord a => Ord [(String, Either (a,Int) [Double])]`
  - functions are not instances of `Eq` and `Ord`: `Eq (Int -> Int)` does not hold

Type Class Hierarchy for Numbers

Type Class `Num`

- `Num a` provides basic arithmetic operations
- specification

```
(+)         :: a -> a -> a
(*)         :: a -> a -> a
(-)         :: a -> a -> a
abs         :: a -> a
signum      :: a -> a
fromInteger :: Integer -> a
negate      :: a -> a
```

- minimal complete definition: nearly everything, only `negate` or `(-)` can be dropped
- number literals are available for instances of `Num` class: `4715 :: Num a => a`
- instances: `Int`, `Integer`, `Float`, `Double`

The `Fractional` Class – Division

- definition: `class Num a => Fractional a where ...`
- excerpt of functions
  ```
  (/) :: a -> a -> a
  ```
- used for fractional literals: `5.72 :: Fractional a => a`
- instances: `Float`, `Double`

The `Integral` Class – Division with Remainder

- definition: `class (Num a, Ord a) => Integral a where ...`
- excerpt of functions
  ```
  toInteger :: a -> Integer
  div       :: a -> a -> a
  mod       :: a -> a -> a
  ```
- instances: `Int`, `Integer`

Different behaviour when dividing by 0

- check: ``1 `div` 0``, `1 / 0`, `1 / (-0)`, `0 == -0`

The `RealFrac` Class – Truncation

- definition: `class (Real a, Fractional a) => RealFrac a where ...`
- excerpt of functions

```
floor :: Integral b => a -> b
ceiling :: Integral b => a -> b
round :: Integral b => a -> b
```

- instances: `Float`, `Double`

Conversion of Numbers

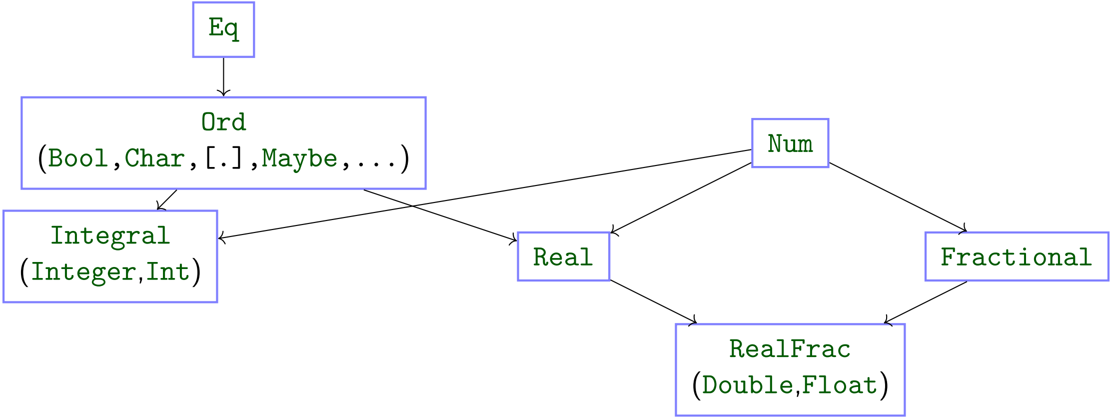
- from integral to arbitrary number type

```
fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral x = fromInteger (toInteger x)
```

- from real fractional numbers to integral numbers

```
fractionalPart :: RealFrac a => a -> a
fractionalPart x = x - fromInteger (floor x)
```

Excerpt of Class Hierarchy



- documentation under
  http://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html

Type Class Instantiation

Instantiating a Type Class

- so far: definitions of type classes, list of existing instantiations
- now: define own instances; syntax is as follows

```
instance (optional type constraints) => TClass (TConstr a1 .. aN) where
  ... -- implementation of functions
```

  where

  - `a1 .. aN` are distinct type variables
  - these may be used in a type constraint
  - the implementation has to provide the implementations for each function `f :: ty` within the definition of `TClass a`
    - however, `f` has to be implemented for type `ty'` which is obtained by replacing `a` by `TConstr a1 .. aN` in `ty`
    - functions `f` that have a default implementation can be omitted
    - whenever a type constraint is used, then the implementation may use the functions of that type class
- writing `deriving TClass` in data type definition triggers generation of default instance; supported for type classes `Eq`, `Ord`, `Show`

Example: Complex Numbers

```haskell
data Complex = Complex Double Double -- real and imaginary part

-- remark: we do not write deriving (Eq, Show),
--   but implement these instances on our own

instance Eq Complex where
  Complex r1 i1 == Complex r2 i2 = r1 == r2 && i1 == i2
  -- for (/=) use default implementation

instance Show Complex where
  show (Complex r i)
    | i == 0 = show r
    | r == 0 = show i ++ "i"
    | i < 0 = show r ++ show i ++ "i"
    | otherwise = show r ++ "+" ++ show i ++ "i"
```

Example: Complex Numbers Continued

```
instance Num Complex where
  Complex r1 i1 + Complex r2 i2 = Complex (r1 + r2) (i1 + i2)
  Complex r1 i1 * Complex r2 i2 =
    Complex (r1 * r2 - i1 * i2) (r1 * i2 + r2 * i1)
  fromInteger x = Complex (fromInteger x) 0
  negate (Complex r i) = Complex (negate r) (negate i)
  abs c = Complex (absComplex c) 0
  signum c@(Complex r i)
    | c == 0 = 0
    | otherwise = Complex (r / a) (i / a)
     where a = absComplex c

-- auxiliary functions must be defined outside
-- the class instantiation
absComplex (Complex r i) = sqrt (r^2 + i^2)
```

Example: Polymorphic Complex Numbers

```haskell
data Complex a = Complex a a -- polymorphic: type a instead of Double

instance Eq a => Eq (Complex a) where
  Complex r1 i1 == Complex r2 i2 = r1 == r2 && i1 == i2
  -- comparing r1 and r2 (i1 and i2) requires equality on type a

-- for Show not only Show a is required, but also Ord a and Num a
instance (Show a, Ord a, Num a) => Show (Complex a) where
  show (Complex r i)
    | i == 0 = show r
    | r == 0 = show i ++ "i"
    | i < 0 = show r ++ show i ++ "i"
    | otherwise = show r ++ "+" ++ show i ++ "i"

instance (Floating a, Eq a) => Num (Complex a) where ...
  -- sqrt :: Floating a => a -> a, Floating a implies Num a
```

Example: Polymorphic Complex Numbers in Action

```
> (Complex 0 1)^2
-1.0

> 2 + 5 :: Complex Float
7.0

> abs (Complex 1 3) :: Complex Float
3.1622777
> abs (Complex 1 3) :: Complex Double
3.1622776601683795

> 2 * Complex 7 2.5
14.0+5.0i

> 2.4 * Complex 7 2.5
error: No instance for (Fractional (Complex Double))
```

Limitations of Type Class Instantiations

- instantiation: `instance ... => TClass (TConstr a1 .. aN)`
- the type variables cannot be replaced by more concrete types
  - example: the following instantiation is not permitted

```
-- show Boolean lists as bit-strings: "011" vs. "[False,True,True]"
instance Show [Bool] where
  show (b : bs) = (if b then '1' else '0') : show bs
  show [] = ""
```

  - workaround via separate function: `showBits :: [Bool] -> String`
- each combination of type class and type can have at most one instance
  - example: the following instantiation is not permitted

```
-- case-insensitive comparison of characters
import Data.Char
instance Ord Char where   -- clashes with existing Ord Char instance
  c <= d = toUpper c <= toUpper d
```

  - workaround: parametrise sorting algorithm, . . . by order instead of using (<=)

Summary

- several type classes are already defined in Prelude
- hierarchy of type classes for numbers
- new type classes can be user defined; content:
  - list of function names with types
  - description of what these functions should do
  - optionally: default implementations for some of the functions
- new type class instantiations can be added,
  where both type class and type can either be user defined or predefined
- for each combination of type class and type, there can be at most one implementation
- conversion between operators and functions: `(+)` ``(25 `div` 3) 2``