Outline of Lecture 10

- Input and output in Haskell
- Type classes: class signatures and instances, interfaces and contexts
- Multiple constraints and derived classes
- Built-in Haskell type classes

Input and output in Haskell

- We consider simplest programs, reading and writing to a terminal
- The described model (solution) forms a foundation for more complex interactions (e.g., with a mail or an operating system)
- The solution relies on the Haskell type `IO a`, describing programs that do some input/output before returning the value of the type `a`
- A number of such programs can be sequenced by the means of the `do` construct

Input and output: problems in functional programming

- Functional program consists of a number of definitions, associating a fixed value with the variable/identifier name
- How to implement input/output in such a programming style?
- One approach (tried in Standard ML and F#) is to include operations/special identifiers like

```
inputInt :: Integer
```

  whose effect is to read an integer from the input. The read value becomes the value of `inputInt`
- How to interpret then the following definition?

$$inputDiff = inputInt - inputInt$$

Input and output: problems in functional programming

$$inputDiff = inputInt - inputInt$$

- Since the values of the first and second occurrences of `inputInt` may be different, evaluation of such a definition breaks the main principle of functional programming stating that an identifier/variable name always stands for a fixed value
- Moreover, the problem propagates in all other definitions relying on `inputDiff`, like

```
funny n = InputInt + n
```

- Such mutability of definitions made I/O quite an issue for functional programming

Input and output: Haskell solution

- A part of the monadic approach (more details later)
- The solution relies on the Haskell type `IO a`, describing actions that do some input/output (or any effects beyond evaluating function or expression) before returning the value of the type `a`
- The type `IO a` contains all I/O actions of the type `a` (i.e., returning, after doing some I/O, the value of the type `a`)
- Such I/O actions are usually done in sequence (read something, calculate next, return some output)
- Haskell provides a small *imperative* language (`do` notation) to sequence such actions

Reading input

- Basic I/O commands (part of Prelude)
- The built-in operation of reading a line from input:

```
getLine :: IO String
```

- Similarly, the operation of reading a single character from input:

```
getChar :: IO Char
```

- In GHCI, executing such commands is delayed until the respective input is supplied

Writing strings into output

- The built-in operation of putting a string to output:

```
putStr :: String -> IO ()
```

- Here `()` represents the Haskell type containing one element (also denoted `()`). Used in the cases to indicate that nothing specific should be returned (similar to `void`). Here, nothing is to be returned back to Haskell after IO actions

- Using this, we can write our "Hello, World!" program in Haskell:

```
helloWorld :: IO ()
helloWorld = putStr "Hello, World!"
```

Writing values in general (printing)

- Printing can be implemented as follows (very close to the actual definition of `print`):

```
myprint ::  Show a => a -> IO ()
myprint s = putStrLn (show s)
```

  where `putStrLn` is defined as

```
putStrLn ::  String -> IO ()
putStrLn st = putStr (st ++ "\n")
```

- Returning a value, without any actual I/O (by the built-in command `return`):

```
return ::  a -> IO a
```

  Return nothing: `return ()`

The Main program

- If we compile a Haskell project using GHC, then it produces executable program, which runs a function :

  `main ::  IO t`

  for some type `t`
- Often, nothing is returned:

```haskell
main ::  IO ()
main = putStrLn "Hello, World!"
```

- By default, the `main` program is expected to be in the `Main` module
- Compiling and running the main program (in the module helloworld.hs):

```
> ghc --make helloworld
> ./helloworld
```

The do notation

- The do notation is used to build IO programs from those and similar primitives we had so far
- In general, it supports sequencing simple IO programs (i.e., ”glue together” several IO actions into one)
- The do notation also allows to capture (name) the values returned by IO actions
- This makes do expression appear like a simple imperative program, containing a sequence of commands and assignments

The do notation (cont.)

- Combining inputs and outputs:

```
read2lines ::  IO ()
read2lines = do
    getLine
    getLine
    putStr "Two lines read."
    putStr "\n"
```

To put several IO actions in one line, use ";"

The do notation (cont.)

- Capturing the read values:

```
reverse2lines ::  IO ()
reverse2lines = do
    line1 <- getLine
    line2 <- getLine
    putStrLn (reverse line2)
    putStrLn (reverse line1)
```

Similar to variable assignments, however, each 'var <- ' creates a new variable var. Therefore, a single assignment, not updatable assigment

The do notation (cont.)

- Local definitions in a do expression:

```
reverse2lines ::  IO ()
reverse2lines = do
    line1 <- getLine
    line2 <- getLine
    let rev1 = reverse line1
    let rev2 = reverse line2
    putStrLn rev2
    putStrLn rev1
```

Using let constructs to introduce local identifiers

Loops and recursion

- Looping is achieved via recursion within the `do` construct:

```
copy ::  IO ()
copy = do
    line <- getLine
    putStrLn line
    copy
```

Running `copy` within GHCI $\Rightarrow$ looping forever; it can be interrupted by Ctrl-C

Loops and recursion (cont.)

- We can control the number of lines by passing the number as a parameter:

```
copyN ::  Integer -> IO ()
copyN n =
    if n <= 0 then
      return ()
    else do
      line <- getLine
      putStrLn line
      copyN (n-1)
```

Similar to while loop (only by recursion)

Loops and recursion (cont.)

- We can also terminate the loop by checking a condition on data:

```
copyEmpty ::  IO ()
copyEmpty = do
    line <- getLine
    if line == "" then
      return ()
    else do
      putStrLn line
      copyEmpty
```

Note: embedded `do` constructs; Anywhere we need to sequence IO actions, the `do` constructs are used

Overloading revisited

- Haskell has two kinds of functions working over more than one type: polymorphic and overloaded
- A polymorphic function has a single definition which works over all its types
- Overloaded functions can be used for a variety of types, but with different definitions at the different types
- What are benefits of overloading?

Overloading (cont.)

- Suppose there is no overloading $\Rightarrow$ need to write functions like the one checking that an element belongs to a boolean list:

```
elemBool ::  Bool -> [Bool] -> Bool
elemBool x [] = False)
elemBool x (y:ys) =
    = (x ==Bool y) || elemBool x ys
```

- Similarly, a different function `elemInt` with $==_{Int}$ instead of $==_{Bool}$
- One possible solution: a generic function like:

```
elemGen ::  (a->a->Bool) -> a -> [a] -> Bool
```

Disadvantages: always to use the extra functional parameter, which is not necessarily an instance of (==)

Overloading (cont.)

- What we need is a definition of the kind

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

  where the type a is restricted to only those types that have an equality operation defined

- **Advantage**: the same definition can be reused over a collection of types

- **Advantage**: it much easier to read == than $==_{Int}$, $==_{Float}$, $==_{Char}$ and so on

- In Haskell, this is realised via the type class mechanism

Introducing type classes

- Intuitively, we can understand *typeclasses* as **interfaces** to data that can work over multiple types
- Typeclasses also provide *constrained polymorphism*, by defining **signature** which has to be implemented for a type to belong to the class
- Typeclasses allow us to generalise over a set of types in order to define and execute a standard set of features for those types. Examples of the pre-defined Haskell classes: `Eq`, `Ord`, `Num`, `Show`, `Enum`, `Bounded`, `...`
- Members of a type class are called its instances. A type is made into an instance by giving an implementation of the interface in an **instance declaration**

Equality typeclass Eq

- Let look at the simple definition of the equality type class, `Eq`

```
Prelude> :info Eq
class Eq a where
  (==) ::  a -> a -> Bool
  (/=) ::  a -> a -> Bool
...
```

- Essentially, we specify here an **interface** or **signature** which has to be implemented for a type to belong to the class
- To declare an instance of the class, there is the **minimal definition** requirement. For `Eq`, it is either providing a concrete definition of `(==)` or `(/=)`

Functions that use equality

- Let us consider the function

```
allEqual ::  Int -> Int -> Int -> Bool
allEqual m n p = (m==n) && (n==p)
```

- There is nothing that makes it specific to integers. It just compares three values for equality

- It can be generalised over all the types that have equality:

```
allEqual ::  Eq a => a -> a -> a -> Bool
allEqual m n p = (m==n) && (n==p)
```

- The part before => is called **context**. It restricts the polymorphic type a to a specific type class

Equality typeclass Eq (cont.)

- There are many built-in instances of `Eq` (listed by `:info Eq`):

```
...
  instance Eq a => Eq [a]
  instance Eq Int
  instance Eq Float
  instance Eq Double
  instance Eq Char
  instance Eq Bool
  instance (Eq a, Eq b) => Eq (a,b)
  instance Eq Integer
  ...
```

- To declare a composite type as a typeclass instance, appropriate typeclass constraints may be needed for its constituent members, e.g., `Eq a => Eq [a]`

Breaking the type class constraint

- Comparing three functions:

```
suc ::  Integer -> Integer
suc = (+1)

> allEqual suc suc suc
```

- Error message:

```
No instance for Eq (Integer -> Integer) ...
Possible fix:  add an instance declaration for
Eq (Integer -> Integer)
```

- By default, no `Eq` instance exists for the function type `Integer -> Integer`

Checking existing instances for a concrete type

- We can always check what typeclasses a concrete type already belongs to:

```
Prelude> :info Bool
data Bool = False | True
instance Bounded Bool
instance Enum Bool
instance Eq Bool
instance Ord Bool
instance Read Bool
instance Show Bool
```

- We can introduce new typeclasses and then add specific type instances into them

Checking existing instances for a concrete type (cont.)

- Another example:

```
Prelude> :info (,)
data (,) a b = (,) a b
instance (Bounded a, Bounded b) => Bounded (a, b)
instance (Eq a, Eq b) => Eq (a, b)
instance (Ord a, Ord b) => Ord (a, b)
instance (Show a, Show b) => Show (a, b)
...
```

- Note that `Eq` (or `Ord`, `Bounded`, `Show`, `...`) instance of `(a,b)` relies on `Eq` instances of `a` and `b`. It is because of the standard definition of == for 2-tuples:

```
(==) (a,b) (c,d) = (a==c) && (b==d)
```

Adding a new datatype to a typeclass

- We can do that by declaring a new instance (with the **instance** block) and providing the minimal definition(s) for implemented functions

```
data DayOfWeek =
  Mon | Tue | Wed | Thu | Fri | Sat | Sun
data Date = Date DayOfWeek Int

instance Eq DayOfWeek where
  (==) Mon Mon = True
  (==) Tue Tue = True
  (==) Wed Wed = True
  (==) Thu Thu = True
  (==) Fri Fri = True
  (==) Sat Sat = True
  (==) Sun Sun = True
  (==) _ _ = False
```

Adding a new datatype to a typeclass (cont.)

- And the same for `Date`

```
instance Eq Date where
  (==) (Date wday mday) (Date wday' mday') =
    wday == wday' && mday == mday'
```

- Here two different definitions of == are used: for `DayOfWeek` and `Int`
- Standard instance implementations (for `Eq`, `Ord`, `Enum`, `Show`) can be automatically created for datatypes by using the keyword **deriving (...)**. More about that later

Introducing our own type classes

- For example, let us declare our type class, `Info`

```
class Info a where
  examples ::  [a]
  size ::  a -> Int
```

- To be in the defined class, a type must implement two interface bindings:
  - the `examples` list – a list of representatives examples,
  - the `size` function, returning a measure of size of the argument.
- How are types made instances of such a class?

Defining instances of a class

- As shown before, we can declare an instance together with definitions of the necessary interface functions. For example:

```
instance Eq Bool where
  (==) True True = True
  (==) False False = True
  (==) _ _ = False
```

- For our `Info` class:

```
instance Info Char where
  examples = ['a','A','z','Z','0','9']
  size _ = 1

instance Info Shape where
  examples = [Circle 3.0, Rectangle 45.1 17.9]
  size = round . area
```

Instances and contexts

- We can rely on the type class information when building instances for composite types, e.g., when making [a] an instance of `Info`, in which the context `Info a` appears to constrain the type a:

```
instance Info a => Info [a] where
  examples = [[x], x <- examples] ++
    [[x,y], x <- examples, y <- examples]
  size = foldr (+) 1. map size
```

- Note that `examples` and `size` used on the definition right hand sides are those defined for the type a and thus are different from those on the left hand side (no recursive calls here!)

Default definitions

- The actual definition of the Eq class:

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

- The last two lines are default definitions for == and /=
- Defaults are overridden by instance definitions
- For the Eq example above, one given implementation definition in an instance declaration is sufficient (the minimal requirement)

Default definitions (cont.)

- For our Info class:

```
class Info a where
  examples ::  [a]
  size ::  a -> Int
  size _ = 1
```

- Then, for some instances, we can simply have, e.g.,:

```
instance Info Char where
  examples = ['a','A','z','Z','0','9']
```

relying here on the default definition of size

Derived classes

- As functions and instances, classes also can depend upon their constituent types already being in (some other) classes, e.g.,

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) ::  a -> a -> Bool
  max, min ::  a -> a -> a
  compare ::  a -> a -> Ordering
  x <= y  =  (x < y || x ==y)
  x > y  =  y < x
```

where data `Ordering = LT | EQ | GT`

- Therefore, any type belonging to `Ord` must belong to `Eq` first (so that equality could be used for comparison). Similar to inheritance in OOP

Multiple constraints

- We can have multiple constraints on types in the context part, e.g.,

```
vSort ::  (Ord a,Show a) => [a] -> String
vSort = show . iSort
```

- Multiple constraints can occur in an instance declaration:

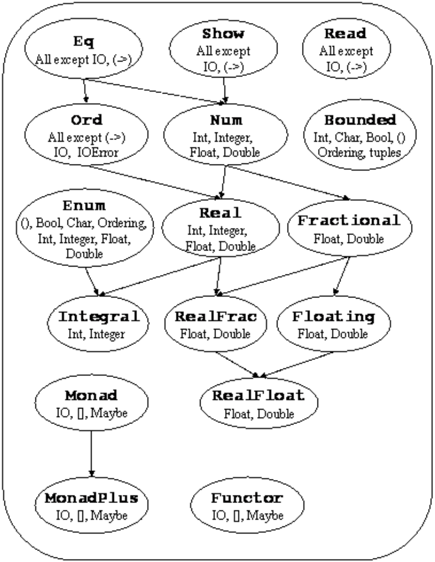
```
instance (Eq a, Eq b) => Eq (a,b) where
  (x,y) == (z,w)  =  x==z && y==w
```

- Multiple constraints can also occur in a class definition:

```
class (Ord a, Show a) => OrdShow a
```

In such a definition, the class inherits all the operations of both `Ord` and `Show`. Like multiple inheritance in OOP

Haskell built-in classes



Haskell built-in classes (cont.)

- The definitions of classes `Eq` and `Ord` – shown before
- Enumeration (`Enum`), supporting enumeration expressions like `[2,4 .. 8]`:

```
class Ord a => Enum a where
  succ, pred ::  a -> a
  toEnum ::  Int -> a
  fromEnum ::  a -> Int
  enumFrom ::  a -> [a]                 -- [n ..  ]
  enumFromThen ::  a -> a -> [a]        -- [n,m ..  ]
  enumFromTo ::  a -> a -> [a]          -- [n ..  m]
  enumFromTo ::  a -> a -> a -> [a]     -- [n,n' ..  m]
```

- Not just integers: characters, floating point numbers, etc.

Haskell built-in classes (cont.)

- Bounded types (`Bounded`):

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

`Int, Char, Bool, Ordering, ...` belong to Bounded

- Turning values to strings (`Show`):

```
class Show a where
  showsPrec ::  Int -> a -> String -> String
  show ::  a -> String
  showList ::  [a] -> String -> String
    ...
```

`showsPrec` supports flexible and efficient conversion of large data values, while `showList` handles conversion of lists.

Haskell built-in classes (cont.)

- In most cases, redefining the `show` function is sufficient, leaving the other functions to their default versions:

```
instance Show Bool where
  show True = "True"
  show False = "False"

instance (Show a, Show b) => Show (a,b) where
  show (x,y) =
    "(" ++ show x ++ "," ++ show y ++ ")"
```

In the last example, different `show` function implementations depending on the type

Haskell built-in classes (cont.)

- The base class for all numeric types (`Num`):

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) ::  a -> a -> a
  negate ::  a -> a
  abs, signum ::  a -> a
  fromInteger ::  Integer -> a
  x - y  =  x + negate y
  ...
```

- The integer types belong to the class `Integral`, including such signature functions as:

```
  quot, rem ::  a -> a -> a
  div, mod ::  a -> a -> a
```

giving two variants of integer division

Haskell built-in classes (cont.)

- Numbers with fractional parts belong to the class `Fractional`:

```
class (Num a) => Fractional a where
  (/) ::  a -> a -> a
  recip ::  a -> a
  fromRational ::  Rational -> a
  recip x    =  1 / x
  ...
```

- The class for floating-point numbers (`Floating`), carrying the basic mathematical functions:

```
class (Fractional a) => Floating a where
  pi ::  a
  exp, log, sqrt ::  a -> a
  (**), logBase ::  a -> a -> a
  sin, cos, tan ::  a -> a
  ...
```

Derived instances

- When a new data type is introduced, it comes with facilities for pattern matching but no other pre-defined functions
- It is possible to come up with standard definitions of equality, ordering, `show` and `read` functions for such types

```
data People = Person Name Age
  deriving (Eq,Show)
```

- As a result, the definitions of `==` and `show` are synthesised for this type
- The described mechanism works for all the standard classes
- Of course, we can declare our data type as an instance of a type class ourselves, redefining the functions as we see fit