

Lecture 5. Data types and type classes

Functional Programming 2019/20

Alejandro Serrano



One more note about higher-order functions

- ▶ How can we use functions?
- ▶ Can we pattern match on them?
- ▶ Can we inspect their source code from a higher-order function?



One more note about higher-order functions

- ▶ How can we use functions?
 - ▶ By applying them! That's it!
- ▶ Can we pattern match on them?
 - ▶ No! But we can feed them inputs and pattern match on the results!
- ▶ Can we inspect their source code from a higher-order function?
 - ▶ No! Only their input-output behaviour!



Goal of typed purely functional programming

Lecture 2:

- ▶ data-flow only through function arguments and return values
 - ▶ no hidden data-flow through mutable variables/state
 - ▶ instead: tuples!

Last time:

- ▶ function call and return as only control-flow primitive
 - ▶ no loops, `break`, `continue`, `goto`
 - ▶ instead: higher-order functions!



Goal of typed purely functional programming

Today:

- ▶ (almost) unique types
 - ▶ no inheritance hell
 - ▶ instead of classes + inheritance: variant types!
- ▶ high-level declarative data structures
 - ▶ no explicit reference-based data structures
 - ▶ instead: (immutable) algebraic data types!



Goals for today

- ▶ Define your own algebraic data types:
 - ▶ tuples (recap), variants, and recursive
- ▶ Define your own type classes and instances
- ▶ Understand the difference between parametric and ad-hoc polymorphism
- ▶ Understand the value and limitations of algebraic data types

Chapter 8 (until 8.6) from Hutton's book



In the previous lectures...

... we have only used built-in types!

- ▶ Basic data types
 - ▶ Int, Bool, Char...
- ▶ Compound types parametrized by others
 - ▶ Some with a definite amount of elements, like tuples
 - ▶ Some with an unbound number of them, like lists

It's about time to define our own!



Direction

```
data Direction = North
                | South
                | East
                | West
```

- ▶ data declares a new **data type**
- ▶ The name of the type must start with **Uppercase**
- ▶ Then we have a number of *constructors* separated by |
 - ▶ Each of them also starting by uppercase
 - ▶ The same constructor cannot be used for different types
- ▶ Such a simple data type is called an *enumeration*



Building a list of directions

Each constructor defines a *value* of the data type

```
> :t North  
North :: Direction
```

You can use Direction in the same way as Bool or Int

```
> :t [North, West]  
[North, West] :: [Direction]  
> :t (North, True)  
(North, True) :: (Direction, Bool)
```



Pattern matching over directions

To define a function, you proceed as usual:

1. Define the type

```
directionName :: Direction -> String
```

2. Enumerate the cases

► The cases are each of the constructors

```
directionName North = _
```

```
directionName South = _
```

```
directionName East = _
```

```
directionName West = _
```



Pattern matching over directions

3. Define each of the cases

```
directionName North = "N"
```

```
directionName South = "S"
```

```
directionName East = "E"
```

```
directionName West = "W"
```

```
> map directionName [North, West]  
["N", "W"]
```



Built-in types are just data types

- ▶ Bool is a simple enumeration

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

- ▶ Int and Char can be thought as very long enumerations

```
data Int  = ... | -1 | 0 | 1 | 2 | ...
```

```
data Char = ... | 'A' | 'B' | ...
```

- ▶ The compiler treats these in a special way



Points

Data types may store information within them

```
data Point = Pt Float Float
```

- ▶ The name of the constructor is followed by the list of types of each argument
- ▶ Constructor and type names may overlap

```
data Point = Point Float Float
```



Using points

- ▶ To create a point, we use the name of the constructor followed by the value of each argument

```
> :t Pt 2.0 3.0  
Pt 2.0 3.0 :: Point
```

- ▶ To pattern match, we use the name of the constructor and further matches over the arguments

```
norm :: Point -> Float  
norm (Pt x y) = sqrt (x*x + y*y)
```

- ▶ Do not forget the parentheses!

```
> norm Pt x y = x * x + y * y
```

```
<interactive>:2:6: error:
```

- The constructor 'Pt' should have 2 arguments, but has been given none



Constructors are functions

Each constructor in a data type is a function which build a value of that type given enough arguments

```
> :t North
North :: Direction  -- No arguments
> :t Pt
Pt :: Float -> Float -> Point  -- 2 arguments
```

They can be arguments or results of higher-order functions

```
zipPoint :: [Float] -> [Float] -> [Point]
zipPoint xs ys = map (uncurry Pt) (zip xs ys)
               -- = [Pt x y | (x, y) <- zip xs ys]
```



Time to work!

Define the `uncurry` function:

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```



Time to work!

Define the `uncurry` function:

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```

```
-- Choose your own style
```

```
uncurry f (x, y) = f x y
```

```
uncurry f      = \ (x, y) -> f x y
```



Shapes

A data type may have zero or more *constructors*, each of them holding zero or more *arguments*

```
data Shape = Rectangle Point Float Float
           | Circle      Point Float
           | Triangle    Point Point Point
```



Pattern matching over shapes

The function `perimeter` returns the length of the boundary of a shape

```
perimeter :: Shape -> Float
```



Pattern matching over shapes

The function `perimeter` returns the length of the boundary of a shape

```
perimeter :: Shape -> Float
```

Gentle basic geometry reminder

$$P_{\text{rect}} = 2w + 2h$$

$$P_{\text{circle}} = 2\pi r$$

$$P_{\text{triang}} = \text{dist}(a, b) + \text{dist}(b, c) + \text{dist}(c, a)$$

Try it yourself!



Pattern matching over shapes

Each case starts with a constructor – in uppercase – and matches the arguments

```
area :: Shape -> Float
area (Rectangle _ w h) = w * h
area (Circle _ r)      = pi * r ^ 2
area (Triangle x y z) = sqrt (s*(s-a)*(s-b)*(s-c))
                        -- Heron's formula

where a = distance x y
      b = distance y z
      c = distance x z
      s = (a + b + c) / 2
```

```
distance (Pt u1 u2) (Pt v1 v2)
  = sqrt ((u1-v1)^2+(u2-v2)^2)
```



ADTs versus object-oriented classes

```
abstract class Shape {  
    abstract float area();  
}  
class Rectangle : Shape {  
    public Point corner;  
    public float width, height;  
    public float area() { return width * height; }  
}  
// More for Circle and Triangle
```

- ▶ There is no *inheritance* involved in ADTs
- ▶ Constructors in an ADT are *closed*, but you can always add *new subclasses* in a OO setting
- ▶ Classes bundle *methods*, functions for ADTs are defined *outside* the data type



Nominal versus structural typing

```
data Point = Pt Float Float
```

```
data Vector = Vec Float Float
```

- ▶ These types are *structurally* equal
 - ▶ They have the same number of constructors with the same number and type of arguments
- ▶ But for the Haskell compiler, they are **unrelated**
 - ▶ You cannot use one in place of the other
 - ▶ This is called *nominal* typing

```
> :t norm
```

```
norm :: Point -> Float
```

```
> norm (Vec 2.0 3.0)
```

```
Couldn't match 'Point' with 'Vector'
```



Lists and trees of numbers

Data types may refer to themselves

- ▶ They are called **recursive** data types

```
data IntList  
  = EmptyList | Cons Int IntList
```

```
data IntTree  
  = EmptyTree | Node Int IntTree IntTree
```



Cooking elemList

1. Define the type

```
elemList :: Int -> IntList -> Bool
```

2. Enumerate the cases

- ▶ One equation per constructor

```
elemList x EmptyList = _  
elemList x (Cons y ys) = _
```

3. Define the cases

```
elemList x EmptyList = False  
elemList x (Cons y ys)  
  | x == y           = True  
  | otherwise        = elemList x ys
```



Cooking elemTree

Time to think!

```
elemTree :: Int -> IntTree -> Bool
```



Cooking elemTree

1. Define the type

```
elemTree :: Int -> IntTree -> Bool
```

2. Enumerate the cases

- ▶ Each constructor needs to come with as many variables as arguments in its definition

```
elemTree x EmptyTree = _
```

```
elemTree x (Node y rs ls) = _
```

3. Define the simple (base) cases

```
elemTree x EmptyTree = False
```



Cooking elemTree

4. Define the other (recursive) cases

- ▶ Each recursive appearance of the data type as an argument usually leads to a recursive call in the function

```
elemTree x (Node y rs ls)
  | x == y      = True
  | otherwise = elemTree x rs || elemTree x ls
```

-- Or simpler

```
elemTree x (Node y rs ls)
  = x == y || elemTree x rs || elemTree x ls
```



Cooking treeHeight

The function `treeHeight` computes the height of a tree, that is, the length of the maximum path from the root to an `EmptyTree`.

```
> treeHeight (Node (Node 1 EmptyTree EmptyTree)  
                  EmptyTree)
```

```
2
```

```
> treeHeight EmptyTree
```

```
0
```

Try it yourself!



Tree height and size

- ▶ The tree *height* is the length of the maximum path from the root to an `EmptyTree`.
- ▶ The tree *size* is the number of nodes it has.

Question

Can you write a single higher-order function which can be instantiated to both?



Cooking treeToList

1. Define the type

```
treeToList :: IntTree -> IntList
```

2. Enumerate the cases

```
treeToList EmptyTree = _
```

```
treeToList (Node x ls rs) = _
```

3. Define the simple (base) cases

```
treeToList EmptyTree = EmptyList
```

How do we proceed now?



Cooking treeToList

4. Define the other (recursive) cases

```
treeToList (Node x ls rs)
  = Cons x (concatList ls' rs')
  where ls' = treeToList ls
        rs' = treeToList rs
```

-- Left as an exercise to the audience

```
concatList :: IntList -> IntList
            -> IntList
concatList xs = _
```



Polymorphic data types

We have seen examples of types which are parametric

- ▶ Lists like `[Int]`, `[Bool]`, `[IntTree]`...
- ▶ Tuples `(A, B)`, `(A, B, C)` and so on

Functions over these data types can be polymorphic

- ▶ They work regardless of the parameter of the type

```
(++) :: [a] -> [a] -> [a]
```

```
zip  :: [a] -> [b] -> [(a, b)]
```



Optional values

Maybe T represents a value of type T which might be absent

```
data Maybe a = Nothing
              | Just a
```

- ▶ In the declaration of a polymorphic data type, the name `Maybe` is followed by one or more type variables
 - ▶ Type *variables* start with a lowercase letter
- ▶ The constructors may refer to the type variables in their arguments
 - ▶ In this case, `Just` holds a value of type `a`



Optional values

```
> :t Just True
Maybe Bool
> :t Nothing
Maybe a
```

Note that `Nothing` has a polymorphic type, since there is no information to fix what `a` is



Cooking find

`find p xs` finds the first element in `xs` which satisfies `p`

- ▶ Such an element may not exist
 - ▶ Think of `find even [1,3]`, or `find even []`
- ▶ Other languages resort to `null` or magic `-1` values
- ▶ Haskell always marks a possible absence using `Maybe`

1. Define the type

```
find :: (a -> Bool) -> [a] -> Maybe a
```

2. Enumerate the cases

```
find p [] = _
```

```
find p (x:xs) = _
```



Cooking find

3. Define the simple (base) cases

```
find _ [] = Nothing
```

4. Define the other (recursive) cases

```
find p (x:xs) | p x      = Just x  
              | otherwise = find p xs
```



elem in terms of find

Let me define a small utility function

```
isJust :: Maybe a -> Bool
isJust Nothing = False
isJust (Just _) = True
```

Then we can define elem as a composition of other functions

```
elem :: Eq a => a -> [a] -> Bool
elem x = isJust . find (== x)
```



Trees for any type

We can generalize our `IntTree` data type

- ▶ This is a polymorphic and recursive data type
- ▶ Mind the parentheses around the arguments

```
data Tree a = EmptyTree
            | Node a (Tree a) (Tree a)
```

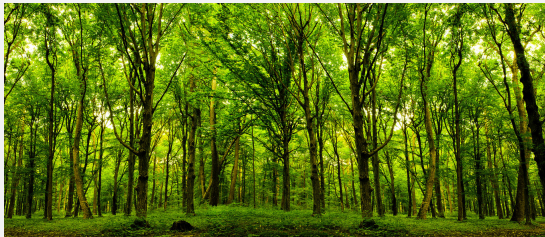


More recipes with trees

Next lecture

Many more operations over trees!

► Including *search* trees



Benefits and downsides of ADTs

- + Immutable
- + Pattern matching and recursion
- Limited to directed, acyclic data types



Type classes



Polymorphism: definitions across many types

Parametric polymorphism - Generics

- ▶ Define once, not inspecting type
- ▶ Works at every instance of parametric data type (infinitely many)

```
reverse :: [a] -> [a]
```



Polymorphism: definitions across many types

Parametric polymorphism - Generics

- ▶ Define once, not inspecting type
- ▶ Works at every instance of parametric data type (infinitely many)

`reverse :: [a] -> [a]`

Ad-hoc polymorphism - Overloading

- ▶ Define many times, inspecting types
- ▶ Works at finitely many types, called *instances* of *type class*, e.g. `Num`, `Eq`

`(+) :: Num a => a -> a -> Bool`

- ▶ **Warning!** Terminology conflict with other languages



Polymorphism

Mixing polymorphism

- Mixing 2 type classes

```
\x -> x == 7  :: (Eq a, Num a) => a -> Bool
```

```
\f -> f 0 == f 1 :: (Eq b, Num a) => (a -> b) -> Bool
```

- Mixing type classes with parametric polymorphism

```
\f x -> f (x + 1) :: Num a -> (a -> b) -> a -> b
```



Class definition

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

- ▶ The name of the type class starts with **U**ppercase
- ▶ We declare a type variable – a in this case – to stand for the overloaded type in the rest of the declaration
- ▶ Each type class defines one or more **methods** which must be implemented for each instance
 - ▶ We do *not* write the constraint in the methods



Missing instances

```
> Pt 2.0 3.0 == Pt 2.0 3.0
```

```
<interactive>:2:1: error:
```

- No instance for (Eq Point)
arising from a use of '=='

- ▶ You have to give the instance declaration for your own data types, even for built-in type classes
 - ▶ In some cases, the compiler can write them for you



Instance declarations

```
instance Eq Point where
```

```
Pt x y == Pt u v = x == u && y == v
```

```
Pt x y /= Pt u v = x /= u || y /= v
```

- ▶ Almost like the class declaration, except that
 - ▶ The type variable is substituted by a real type
 - ▶ Instead of method types, you give the implementation

```
> Pt 2.0 3.0 == Pt 2.0 3.0
```

```
True
```



Recursive instances

Type class instances for polymorphic types may depend on their parameters

- ▶ For example, equality of lists, tuples, and trees
- ▶ These requisites are listed in front of the declaration

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

```
instance (Eq a, Eq b) => Eq (a, b) where
    (x, y) == (u, v) = x == u && y == v
```



Overlapping instances

Imagine that I want tuples of `Ints` to work slightly different

```
instance Eq (Int, Int) where  
    (x, y) == (u, v) = x * v == y * u
```

You *cannot* do this! This instance **overlaps** with the other one given for generic tuples



Recursive instances

Write the Eq instance for the Tree data type:

```
data Tree a = EmptyTree  
            | Node a (Tree a) (Tree a)
```



Recursive instances

Write the Eq instance for the Tree data type:

```
data Tree a = EmptyTree
            | Node a (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
    EmptyTree == EmptyTree
        = True
    (Node x1 l1 r1) == (Node x2 l2 r2)
        = x1 == x2 && l1 == l2 && r1 == r2
        ==
        -
    = False
```



Superclasses

A class might demand that other class is implemented

- ▶ We say that such a class has a **superclass**
- ▶ For example, any class with an ordering – `Ord` – has to implement equality – `Eq`

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

instance (Ord a, Ord b) => Ord (a, b) where
  (x, y) < (u, v) | x == u    = y < v
                  | otherwise = x < u
```



The meanings of \Rightarrow

- ▶ In a type, it constraints a polymorphic function
`elem :: Eq a => a -> [a] -> Bool`
- ▶ In a class declaration, it introduces a superclass
`class Eq a => Ord a where ...`
 - ▶ All instances of `Ord` must be instances of `Eq`
- ▶ In an instance declaration, it defines a requisite
`instance Eq a => Eq [a] where ...`
 - ▶ A list `[T]` supports equality only if `T` supports it

Before \Rightarrow you write an *assumption* or *precondition*



Default definitions

We could also write the following instance `Eq Point`

```
instance Eq Pt where
    Pt ... == Pt ... = _    -- as before
    p /= q = not (p == q)
```

In fact, this definition of `(/=)` works for *any* type

- ▶ You can include a *default* definition in `Eq`
- ▶ If an instance does not have an explicit definition for that method, the default one is used

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



Default definitions

- ▶ You could have also defined (\neq) *outside* of the class

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

 - ▶ This definition cannot be overridden in each instance
- ▶ Why do we prefer (\neq) to live in the class?
 - ▶ Performance! For some data types it is cheaper to check for disequality than for equality



Automatic derivation

- ▶ Writing equality checks is boring
 - ▶ Go around all constructors and arguments
- ▶ Writing order checks is even more boring
- ▶ Turning something into a string is also boring

Let the compiler work for you!

```
data Point = Pt Float Float
           deriving (Eq, Ord, Show)
```

Historical note: many of the advances in automatic derivation of type classes were done here at UU



Define your own data types!

Data types in Haskell are simple and cheap to define

- ▶ Introduce one per concept in your program

-- the following definition

```
data Status = Stopped | Running
```

```
data Process = Process ... Status ...
```

-- is better than

```
data Process = Process ... Bool ...
```

-- what does 'True' represent here?

- ▶ Use type classes to share commonalities



Example: scalable things

Both shapes and vector have a notion of *scaling*

- Scale the size or scale the norm

```
class Scalable s where  
  scale :: Float -> s -> s
```



Example: scalable things

Both shapes and vector have a notion of *scaling*

- Scale the size or scale the norm

```
class Scalable s where  
  scale :: Float -> s -> s
```

```
instance Scalable Vector where  
  scale s v@(Vec x y) = Vec (n*x) (n*y)  
    where n = s / norm v
```

```
instance Scalable Shape where  
  scale s (Rectangle p w h) = Rectangle p (s*w) (s*h)  
  scale s (Circle p r)      = Circle p (s*r)  
  scale s (Triangle x y z)  = ... -- This is hard
```



Generic functions for scalable things

- ▶ Some functions now work over any scalable thing

```
double :: Scalable s => s -> s  
double = scale 2.0
```

- ▶ We may generic instances for composed scalables

```
instance Scalable s => Scalable [s] where  
    scale s = map (scale s)
```



Group work time!

1. Think about a generic notion (like scaling)
2. Define a type class with the least primitive operations
3. Think of instances for that type class
4. Think of derived operations using the type class



Overloaded syntax



Numeric constants' weird type

What is going on?

```
> :t 3
```

```
3 :: Num t => t
```

Numeric constants can be turned into any `Num` type

```
> 3 :: Integer
```

```
3
```

```
> 3 :: Float
```

```
3.0
```

```
> 3 :: Rational -- Type of fractions
```

```
3 % 1 -- Numerator % Denominator
```



Range syntax

The range syntax `[n .. m]` is a shorthand for

```
enumFromTo n m
```

`enumFromTo` lives in the class `Enum`

► `Bool` and `Char` are instances, among others

```
> ['a' .. 'z']
```

```
"abcdefghijklmnopqrstuvwxyz"
```



More range syntax

```
enumFrom      :: a -> [a]
enumFromThenTo :: a -> a -> a -> [a]
```

- ▶ `enumFrom` does not specify a bound for the range
 - ▶ The list is possibly infinite

```
> take 5 [1 ..]
[1,2,3,4,5]
```

- ▶ `enumFromThenTo` generates a list where each pair of adjacent elements has the same distance

```
> [1.0, 1.2 .. 2.0]
[1.0,1.2,1.4,1.5999999999999999,
 1.7999999999999998,1.9999999999999998]
```



Deriving Enum

enumFromTo can be automatically derived for enumerations

- ▶ Data types without data in their constructors

```
data Direction = North | South | East | West
               deriving (Eq, Ord, Show, Enum)
```

```
> [South .. West]
[South, East, West]
```

