Lecture 5. Data types and type classes

Functional Programming 2017/18

Alejandro Serrano
Goals

- Define your own data types
  - Simple, parametric and recursive
- Define your own type classes and instances

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

```haskell
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

   \[
   \text{directionName} :: \text{Direction} \rightarrow \text{String}
   \]

2. Enumerate the cases

   ▶ The cases are each of the constructors

   \[
   \text{directionName North} = _ \\
   \text{directionName South} = _ \\
   \text{directionName East} = _ \\
   \text{directionName West} = _
   \]
3. Define each of the cases

```plaintext
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
  
  ```haskell
  data Bool = False | True
  ```

- **Int** and **Char** can be thought as very long enumerations
  
  ```haskell
  data Int  = ... | -1 | 0 | 1 | 2 | ...
  data Char = ... | 'A' | 'B' | ...
  ```

- The compiler treats these in a special way
Data types may store information within them

```haskell
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

```haskell
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]
Shapes

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

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

We call these *algebraic data types*, or *ADTs*. 
Pattern matching over shapes

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

\[
\text{area} :: \text{Shape} \rightarrow \text{Float} \\
\text{area} (\text{Rectangle \_ w h}) = w \times h \\
\text{area} (\text{Circle \_ r}) = \pi \times r \times r \\
\text{area} (\text{Triangle x y z}) = \sqrt{s*(s-a)*(s-b)*(s-c))} \\
\text{\quad -- Heron's formula}
\]

\[
\text{where} \quad a = \text{distance x y} \\
\text{b} = \text{distance y z} \\
\text{c} = \text{distance x z} \\
\text{s} = (a + b + c) / 2
\]

\[
\text{distance (Pt u1 u2) (Pt v1 v2)} \\
\quad = \sqrt{(u1-v1)*(u1-v1)+(u2-v2)*(u2-v2))}
\]
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

```haskell
data Point = Pt Float Float
data Vector = Vec Float 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 :: Vector -> Float
> norm (Pt 2.0 3.0)
Couldn't match ‘Vector’ with ‘Point’
Lists and trees of numbers

Data types may refer to themselves

- They are called **recursive** data types

```haskell
data ListOfNumbers
  = EmptyList | OneMore Int ListOfNumbers

data TreeOfNumbers
  = EmptyTree | Node Int TreeOfNumbers TreeOfNumbers```

Cooking elemList

1. Define the type

   `elemList :: Int -> ListOfNumbers -> Bool`

2. Enumerate the cases
   - One equation per constructor

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

3. Define the cases

   `elemList x EmptyList = False`
   `elemList x (OneMore y ys)`
   | `x == y = True`
   | otherwise = `elemList x ys`
Cooking `elemTree`

1. Define the type

   ```
   elemTree :: Int -> TreeOfNumbers -> Bool
   ```

2. Enumerate the cases
   - Each constructor needs to come with as many variables as arguments in its definition

   ```
   elemList x EmptyTree = _
   elemList x (Node y rs ls) = _
   ```

3. Define the simple (base) cases

   ```
   elemList x EmptyTree = False
   ```
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

\[
\text{elemList } x \ (\text{Node } y \ rs \ ls) \\
| \ x == y \ = \ True \\
| \ \text{otherwise} = \ \text{elemList } x \ rs \ || \ \text{elemList } x \ ls
\]

-- Or simpler

\[
\text{elemList } x \ (\text{Node } y \ rs \ ls) \\
= \ x == y \ || \ \text{elemList } x \ rs \ || \ \text{elemList } x \ ls
\]
Cooking `treeToList`

1. Define the type

   ```plaintext
treeToList :: TreeOfNumbers ->ListOfNumbers
```

2. Enumerate the cases

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

3. Define the simple (base) cases

   ```plaintext
treeToList EmptyTree = EmptyList
```
4. Define the other (recursive) cases

\[
\text{treeToList} (\text{Node x ls rs}) = \text{OneMore} x (\text{concatList ls'} rs')
\]
where \(ls' = \text{treeToList} \hspace{0.5cm} ls\)
\(rs' = \text{treeToList} \hspace{0.5cm} rs\)

-- Left as an exercise to the audience
\[
\text{concatList} :: \text{ListOfNumbers} \rightarrow \text{ListOfNumbers} \rightarrow \text{ListOfNumbers}
\]
\[
\text{concatList} \hspace{0.5cm} xs = _
\]
Polymorphic data types

We have seen examples of types which are parametric

- Lists like [Int], [Bool], [TreeOfNumbers]...
- 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
   
   ```haskell
   find :: (a -> Bool) -> [a] -> Maybe a
   ```

2. Enumerate the cases
   
   ```haskell
   find p [] = _
   find p (x:xs) = _
   ```
3. Define the simple (base) cases

   \[
   \text{find } _\bot \ [ ] = \text{Nothing}
   \]

4. Define the other (recursive) cases

   \[
   \text{find } p \ (x:xs) \ | \ p \ x \ = \ Just \ x \\
   \hspace{1cm} | \ \text{otherwise} \ = \ \text{find } p \ xs
   \]
Let me define a small utility function

\[
\text{isJust} :: \text{Maybe } a \rightarrow \text{Bool}
\]
\[
\text{isJust } \text{Nothing} = \text{False}
\]
\[
\text{isJust} (\text{Just } _) = \text{True}
\]

Then we can define \text{elem} as a composition of other functions

\[
\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}
\]
\[
\text{elem } x = \text{isJust . find } (== x)
\]
Trees for any type

We can generalize our TreeOfNumbers data type

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

```haskell
data Tree a = Leaf
             | Node a (Tree a) (Tree a)
```
More recipes with trees

Next lecture

Many more operations over trees!

- Including *search* trees
Type classes
Oveloaded types

From previous lectures...

Some functions work uniformly for all types

reverse :: [a] -> [a]

But others require the type to satisfy a constraint

elem :: Eq a => a -> [a] -> Bool
(+) :: Num a => a -> a -> a

- Eq and Num are called **(type) classes**
- Each type which satisfies the constraint is an **instance**
  - Int is an instance of class Eq
- **Warning!** Terminology conflict with other languages
Class definition

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

- The name of the type class starts with Uppercase
- 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

```haskell
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

```haskell
> Pt 2.0 3.0 == Pt 2.0 3.0
True
```
Instance signatures

It is useful to write the specialized type for the instance in the declaration

```haskell
instance Eq Point where
  (==) :: Point -> Point -> Bool
  Pt x y == Pt u v = x == u && y == v
  (/=) :: Point -> Point -> Bool
  Pt x y /= Pt u v = x /= u || y /= v
```

The Haskell standard does not allow this

- But you can do this if you write at the top of the file

  ```haskell
  {-# language InstanceSigs #-}
  ```
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

\[
\text{instance } \text{Eq } a \Rightarrow \text{Eq } [a] \text{ where } \\
[\text{[]}] & \text{==} [\text{[]}] & \text{=} \text{True} \\
[\text{[]}] & \text{==} \text{_} & \text{=} \text{False} \\
\text{_} & \text{==} [\text{[]}] & \text{=} \text{False} \\
(\text{x:xs}) & \text{==} (\text{y:ys}) & \text{=} \text{x} \text{==} \text{y} \text{&&} \text{xs} \text{==} \text{ys}
\]

\[
\text{instance } (\text{Eq } a, \text{Eq } b) \Rightarrow \text{Eq } (a, b) \text{ where } \\
(\text{x, y}) & \text{==} (\text{u, v}) & \text{=} \text{x} \text{==} \text{u} \text{&&} \text{y} \text{==} \text{v}
\]
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
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

```haskell
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 =>

- In a type, it constraints a polymorphic function
  \[ \text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \]

- In a class declaration, it introduces a superclass
  \[ \text{class Eq } a \Rightarrow \text{Ord } a \text{ where } \ldots \]
  - All instances of \text{Ord} must be instances of \text{Eq}
  
- In an instance declaration, it defines a requisite
  \[ \text{instance Eq } a \Rightarrow \text{Eq } [a] \text{ where } \ldots \]
  - A list \([T]\) supports equality only if \(T\) supports it

Before => 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 a 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 (/=) *outside* of the class
  
  \[\text{(=/=) :: Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}\]
  
  \[x \neq y = \text{not } (x == y)\]

- This definition cannot be overridden in each instance

- Why do we prefer (/=) 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!

```haskell
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

```haskell
-- 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

```haskell
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

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

- We may generic instances for composed scalables
  
  ```haskell
  instance Scalable s => Scalable [s] where
  scale s = map (scale s)
  ```
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

\[
\text{enumFrom} :: a \to [a] \\
\text{enumFromThenTo} :: a \to a \to a \to [a]
\]

- \text{enumFrom} does not specify a bound for the range
  - The list is possibly infinite

\[
> \text{take} \ 5 \ [1 ..] \\
[1, 2, 3, 4, 5]
\]

- \text{enumFromThenTo} generates a list where each pair of adjacent elements has the same distance

\[
> [1.0, 1.2 .. 2.0] \\
[1.0, 1.2, 1.4, 1.5999999999999999, 1.7999999999999998, 1.9999999999999998]
\]
enumFromTo can be automatically derived for enumerations

- Data types without data in their constructors

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

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