

Higher Order Functions and Type Classes

Today

How is functional programming distinctive?

Higher-order functions.

Type classes.

2

Haskell and Java

Some of what we've already done can also be done in Java:

Examples:

- recursion,
- lists,
- alternatives (guards),
- sequencing, ...

3

What is distinctive about Haskell?

Other things are distinctive to Haskell (+ other FLs).

Pattern matching,

```
sft (x:y:zs) = x+y  
sft _       = 0
```

list comprehensions,

```
[ 2*n | n<-ns, n>0 ]
```

polymorphism,

```
concat :: [[a]] -> [a]
```

data-directed programming (e.g. `Index.hs`), ...

4

Polymorphism and type safety

Haskell

```
concat :: [[a]] -> [a]
```

Each list is of a given type ...

... but `concat` can join lists of any (single) type.

This is checked when you `write` the program.

Which is better: compile- or run-time?

5

Java

Lists of `Objects`: can join

lists of `Apples` and `Oranges`,

... but `run-time` error when you cast `Apple` → `Orange`

Two other distinctive features

Functions can be passed around as data ... functions can be arguments or results of other functions, known as `higher-order functions`.

Function names can be overloaded: similar to Java interfaces. This uses the `type class` mechanism.

6

Higher Order Functions

Apply to all elements of a list: `map`

Apply `f` to every element of a list:

```
map f [] = []  
map f (x:xs) = f x : map f xs
```

or the list comprehension solution ...

```
map f xs = [ f x | x<-xs ]
```

8

Examples using `map`

Flip in a vertical mirror

```
flipV pic = map reverse pic
```

Find the people in a database

```
people dBase = nub (map fst dBase)
```

Check whether all numbers in a list are even ...

```
checkEven xs = and (map isEven xs)
```

9

The type of `map`

Look at an example ...

```
map isEven [1,5,6] = [False, False, True]
```

function from
`Int` to `Bool`

list of
`Int`

list of
`Bool`

```
map :: (Int -> Bool) -> [Int] -> [Bool]
```

```
map :: (a -> b) -> [a] -> [b]
```

10

Selecting some elements: `filter`

Select those elements for which `p` holds:

```
filter p [] = []  
filter p (x:xs)  
  | p x      = x : filter p xs  
  | otherwise = filter p xs
```

Or via list comprehension:

```
filter p xs = [x | x<-xs, p x]
```

11

The type of `filter`

Look at an example ...

```
filter isEven [2,5,6] = [2,6]
```

function from
`Int` to `Bool`

list of
`Int`

list of
`Int`

```
filter :: (Int -> Bool) -> [Int] -> [Int]
```

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

12

Zippping and mapping along 2 lists

```
sideBySide [l1,l2,l3] [r1,r2,r3]
  = [ l1++r1, l2++r2, l3++r3 ]

zipWith f (x:xs) (y:ys)
  = f x y : zipWith f xs ys

zipWith f xs ys = []

zipWith :: (a->b->c) -> [a] -> [b] -> [c]
sideBySide = zipWith (++)
```

13

Functions as values and results

```
twice :: (a -> a) -> (a -> a)
twice f = (f . f)

double :: Int -> Int
double n = n+n

(twice double) 2 ~> (double.double) 2 ~>
double (double 2) ~> double (2+2) ~>
(2+2) + (2+2) ~> 8
```

14

Lambda notation: anonymous functions

Example: `\n -> n+n` is the doubling function

Notation:

Parameter name: `n`

Return value: `n+n`

General format:

`\ parameter-name -> return-value`

15

Lambda notation: Some examples

Adding 2 to every element of a list:

```
map (\x -> x+2) [1,2]
```

Removing negative numbers:

```
filter (\x -> 0 <= x) [1,-2,3]
```

16

Partial application

```
multiply :: Int -> Int -> Int
multiply x y = x*y
```

A function for multiplying by 2:

```
multiply 2 :: Int -> Int
```

A function for multiplying every element by 2:

```
map (multiply 2) :: [Int] -> [Int]
```

17

A note on function types

```
t1 -> t2 -> t3
is a shorthand for
t1 -> (t2 -> t3)
```

Therefore `f :: t1 -> (t2 -> t3)`
applied to `a :: t1`
yields `f a :: t2 -> t3`

18

A shorthand: operator sections

(+2) a function for adding 2
(0<) a test for positivity

For any infix operator *:

```
(*e) = (\x -> x*e)
(e*) = (\x -> e*x)
```

19

Highly generic functions

Polymorphism: works for a whole family of types

```
concat :: [[a]] -> [a]
```

Higher order: works for a whole family of functions

```
map :: (a -> b) -> [a] -> [b]
```

Supports reuse in many (unforeseen) contexts.

20

Type Classes

Equality and other functions

Equality is defined over many types

```
Float, Int, (Int, Float)
```

but not all types

```
Float->Int, IO String
```

Also it has different definitions at different types

```
2==3      -- built in

(x1,y1) == (x2,y2)
= (x1==x2) && (y1==y2)
```

22

Type classes

There is a Haskell type class `Eq` to which all the types with equality belong, defined by ...

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

For `t` to belong to `Eq` there must be a function of type `t -> t -> Bool`

```
instance Eq Bool where
  ... == over Bool defined in here ...
```

23

The type of equality

```
(==) :: Int -> Int -> Bool
```

not general enough.

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

too general.

```
(==) :: Eq a => a -> a -> Bool
```

it has type `a->a->Bool` ...
... provided that `a` belongs to the type class `Eq`

24

Functions using equality

We've seen various generic functions that use `==`.

```
nub :: Eq a => [a] -> [a]
```

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

```
books :: Eq a => [(a,b)] -> a -> [b]
```

```
books ps z  
= [ y | (x,y)<-ps, x==z ]
```

25

Other built-in type classes (Sec 12.4)

Ord: carry the ordering relations like `<`; extends **Eq**.

```
isort :: Ord a => [a] -> [a]
```

Num: numeric operations like `+`, `*`,

```
add3All :: Num a => [a] -> [a]
```

```
add3All xs = map (+3) xs
```

Fractional, **Floating**: carry division, `pi` etc.

Show: can turn into a **String**

26

Type classes and Java

Haskell type classes resemble Java **interfaces**.

Major difference: **separate** instance declarations from the definition of the type itself...

... new types can support old interfaces, but also old types can conform to new interfaces.

A Haskell type class is **not** a Haskell type (a Java interface is a Java type)

27