Functional Programming

Lecture 6 — Higher order functions

Pedro Vasconcelos
DCC/FCUP

2025

Higher-order functions

A function is higher-order if it has a function argument or a function result.

Example: the first argument of map is a function, hence it is a higher-order function.

```
ghci> map (^2) [1,2,3,4]
[1,4,9,16]
```

Why are higher-order functions useful?

- They allow *parametrizing computations* by passing operations and not just data
- They allow defining common *patterns of computation* that can be easily re-used
- We can prove *general properties* that hold for higher-order functions regardless of the use
  - e.g. *map* does not change the length of the argument list

This lecture

We will look at some higher-order functions that are predefined in the Prelude:

- `map`
- `filter`
- `takeWhile`, `dropWhile`
- `all`, `any`
- `foldr`, `foldl`
- `(.)` (composition)

The *map* function

The *map* function applies a function to every element in a list.

```
map :: (a → b) → [a] → [b]
```

Examples

```
ghci> map (+1) [1,3,5,7]
[2,4,6,8]

ghci> map isLower "Hello!"
[False,True,True,True,True,False]
```

The *map* function (cont.)

We could define *map* using a list comprehension:

```
map f xs = [f x | x←xs]
```

We can also use recursion:

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

The recursive form is useful if you want to prove properties using mathematical induction.

The *filter* function

The *filter* function selects elements from a list that satisfy a predicate (i.e. a boolean function).

```
filter :: (a → Bool) → [a] → [a]
```

Examples

```
ghci> filter (>5) [1..10]
[6,7,8,9,10]

ghci> filter (\n → n `mod` 2==0) [1..10]
[2,4,6,8,10]

ghci> filter isLower "Hello, world!"
"elloworld"
```

The *filter* function (cont.)

We can define *filter* using a list comprehension:

```
filter p xs = [x | x←xs, p x]
```

Again we can also define *filter* using recursion:

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

The *takeWhile* and *dropWhile* functions

Both functions have the same most general type:

```
takeWhile, dropWhile :: (a → Bool) → [a] → [a]
```

takeWhile selects the largest prefix of list whose elements satisfy a predicate

dropWhile removes the largest prefix from list whose elements satisfy a predicate

The *takeWhile* and *dropWhile* functions (cont.)

Examples

```
ghci> takeWhile isLetter "Hello, world!"
"Hello"

ghci> dropWhile isLetter "Hello, world!"
", world!"

ghci> takeWhile (\n → n*n<10) [1..5]
[1,2,3]

ghci> dropWhile (\n → n*n<10) [1..5]
[4,5]
```

The *takeWhile* and *dropWhile* functions (cont.)

We can define both functions using recursion (but *not* using list comprehensions).

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p [] = []
takeWhile p (x:xs)
    | p x       = x : takeWhile p xs
    | otherwise = []

dropWhile :: (a → Bool) → [a] → [a]
dropWhile p [] = []
dropWhile p (x:xs)
    | p x       = dropWhile p xs
    | otherwise = x:xs
```

The *all* and *any* functions

all checks if a predicate holds for all elements in a list

any check if a predicate holds for some element in a list

Both functions admin the same most general type:

```
all, any :: (a → Bool) → [a] → Bool
```

The *all* and *any* functions (cont.)

Examples

```
ghci> all (\n → n`mod`2==0) [2,4,6,8]
True

ghci> any (\n → n`mod`2/=0) [2,4,6,8]
False

ghci> all isLower "Hello, world!"
False

ghci> any isLower "Hello, world!"
True
```

The *all* and *any* functions (cont.)

We can define *all* and *any* using *map*, *and*, *or*:

```
-- map :: (a → b) → [a] → [b]
-- and, or :: [Bool] → Bool
all p xs = and (map p xs)
any p xs = or (map p xs)
```

We can also define the functions by recursion:

```
all p []     = True
all p (x:xs) = p x && all p xs

any p []     = False
any p (x:xs) = p x || any p xs
```

(These definitions are efficient because `&&` and `||` are lazily evaluated.)

The *foldr* function

Many recursive functions on lists follow the followng “primitive recursive” pattern:

```
f []     = z
f (x:xs) = x ⊕ f xs
```

Hence, functions *f* transforms

the empty list into *z*;

the non-empty list $x : xs$ by combining the recursive result of *f* on *xs* with *x* using some operation $\oplus$

The *foldr* function (cont.)

Examples

```
sum []     = 0
sum (x:xs) = x + sum xs
```

$z = 0$

$\oplus = +$

```
product []     = 1
product (x:xs) = x * product xs
```

$z = 1$

$\oplus = *$

```
and []     = True
and (x:xs) = x && and xs
```

$z = \texttt{True}$

$\oplus = \texttt{\&\&}$

```
or []     = False
or (x:xs) = x || or xs
```

$z = \texttt{False}$

$\oplus = \texttt{||}$

```
length []     = 0
length (x:xs)= 1 + length xs
```

$z = 0$

$\oplus = \backslash \_\ n \rightarrow 1 + n$

The *foldr* function (cont.)

The higher-order *foldr* function ("fold right") abstract this recursive pattern.

```
sum     = foldr (+) 0

product = foldr (*) 1

and     = foldr (&&) True

or      = foldr (||) False

length  = foldr (\_ n → 1+n) 0
```

The *foldr* function (cont.)

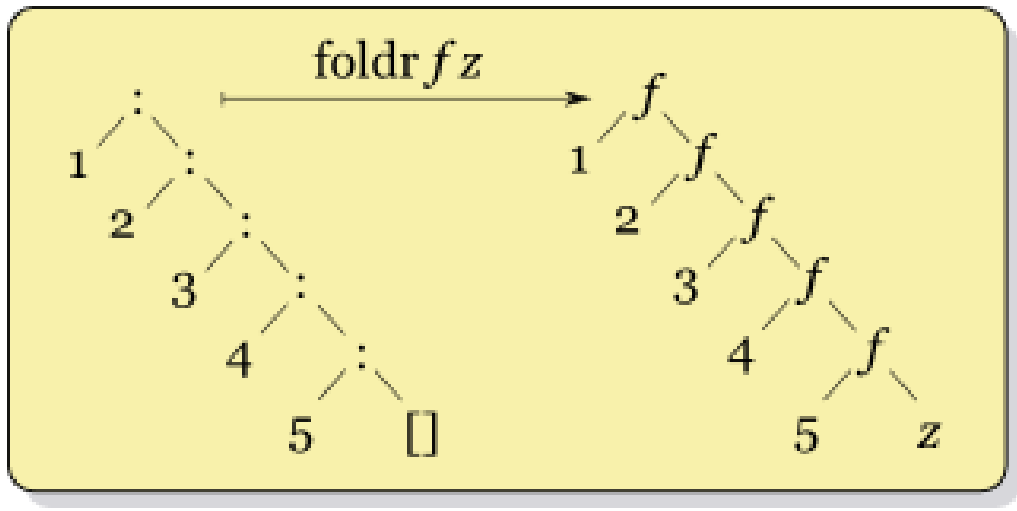
The recursive definition (from the Prelude) expresses this generic pattern.

```
foldr :: (a → b → b) → b → [a] → b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

The *foldr* function (cont.)

However, it is better to visualize *foldr f z* as structural transformation on lists:

- each `:` is transformed into an application of *f*
- the final `[]` is transformed into *z*




```
      foldr f z [1,2,3,4,5]
=     foldr f z (1:2:3:4:5:[])
=  f 1 (f 2 (f 3 (f 4 (f 5 z))))
```

The *foldr* function (cont.)

Example

```
  sum [1,2,3,4]
=
  foldr (+) 0 [1,2,3,4]
=
  foldr (+) 0 (1:(2:(3:(4:[]))))
=
  1+(2+(3+(4+0)))
=
  10
```

The *foldr* function (cont.)

Another example

```
  product [1,2,3,4]
=
  foldr (*) 1 [1,2,3,4]
=
  foldr (*) 1 (1:(2:(3:(4:[]))))
=
  1*(2*(3*(4*1)))
=
  24
```

The *foldl* function

The function *foldr* (*fold right*) reduces a list associating the operation to the right.

$$\mathit{foldr}\ (\oplus)\ e\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus e) \ldots))$$

There is a alternative *foldl* function (*fold left*) that reduces a list associating the operation to the left.

$$\mathit{foldl}\ (\oplus)\ e\ [x_1, x_2, \ldots, x_n] = ((\ldots ((e \oplus x_1) \oplus x_2) \ldots) \oplus x_n)$$

The *foldl* function (cont.)

If the operation $\oplus$ is associative and e is the neutral element, then the result of associating to the left and the right is the same.

```
  foldl (+) 0 [1,2,3,4]
=
  (((0+1)+2)+3)+4
=
  10
```

```
  foldr (+) 0 [1,2,3,4]
=
  1+(2+(3+(4+0)))
=
  10
```

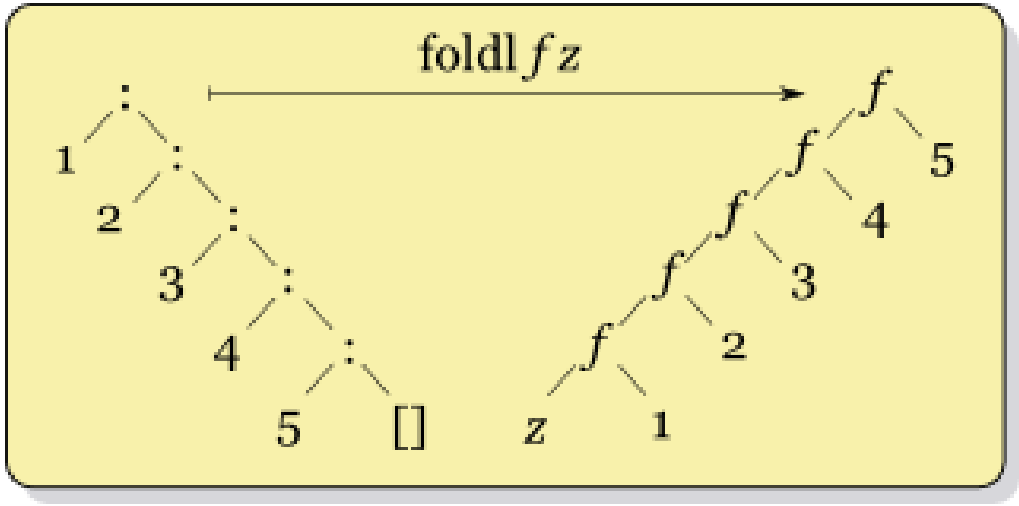
The *foldl* function (cont.)

As with *foldr*, the *foldl* function is defined by recursion:

```
foldl :: (a → b → a) → a → [b] → a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

The *foldl* function (cont.)

But is may be easier to visualize *foldl* as a structural transformation on lists.




```
      foldl f z [1,2,3,4,5]
=     foldl f z (1:2:3:4:5:[])
= f (f (f (f (f z 1) 2) 3) 4) 5
```

Composition

The final higher-order function we will discuss is the *composition* of two functions.

```
(.) :: (b → c) → (a → b) → a → c
f . g = \x → f (g x)
```

Example

```
even :: Int → Bool
even x = x `mod` 2 == 0

odd :: Int → Bool
odd = not . even
```

Composition (cont.)

Composition allow us to simplify nested applications of functions by omitting the parenthesis and the argument.

Example

```
f xs = sum (map (^2) (filter even xs))
```

can also be writen as

```
f = sum . map (^2) . filter even
```

This kind of function definition is called a *point-free* because it omits the argument *xs* (“point”).