Higher-order functions

Lecture 4 of CSE 3100 Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

Lecture plan

- Higher-order functions
- `map`, `filter`, and other functions on lists
- The functions `foldr` and `foldl`

Higher-order functions

The DRY principle of programming

DRY: **Don't Repeat Yourself**

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system*[1]

Higher-order functions are the ultimate expression of DRY, as they allow you to abstract over programming patterns.

[1]from *The pragmatic programmer* by Hunt & Thomas

Higher-order functions

A higher-order function is a function that either takes a function as an argument or returns a function as a result.

The latter is also called a curried function, so the term is mainly used for the former.

Example of a higher-order function

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

> twice (\x -> x*2) 3
12
> twice reverse [1,2,3]
[1,2,3]
```

Quiz question

Question. A function of type `(Bool -> Int) -> Int`

1. …takes a function as its input, which returns an integer as its output
2. …takes a function as its input, which returns a boolean as its output
3. …returns a function as its output, which takes a boolean as its input
4. …returns a function as its output, which takes an integer as its input

Higher-order functions: `curry`, `uncurry`, and `flip`

```
> :t curry
curry :: ((a , b) -> c) -> (a -> b -> c)
> :t uncurry
uncurry :: (a -> b -> c) -> ((a , b) -> c)
> :t flip
flip :: (a -> b -> c) -> (b -> a -> c)
> map (uncurry (+)) [(1,2),(3,4),(5,6)]
[3,7,11]
```

Higher-order function: `($)`

```
> :t ($)
($) :: (a -> b) -> a -> b
```

Question. Why would you ever want to use this?

Higher-order functions on lists

Higher-order function: `map`

```
> :t map
map :: (a -> b) -> [a] -> [b]
> map (\x -> x+1) [1,3,5,7]
[2,4,6,8]
```

`map f xs` corresponds to the list comprehension `[ f x | x <- xs ]`

Quiz question

Question. Which of these equations does NOT hold for all `f :: Int -> Int` and `xs :: [Int]`?

1. `map f (take n xs) == take n (map f xs)`
2. `map f (drop n xs) == drop n (map f xs)`
3. `map f (reverse xs) == reverse (map f xs)`
4. `map f (sort xs) == sort (map f xs)`

Higher-order function: `filter`

```
> :t filter
filter :: (a -> Bool) -> [a] -> [a]
> filter even [1..8]
[2,4,6,8]
```

`filter p xs` corresponds to the list comprehension `[ x | x <- xs , p x ]`

Using `map`/`filter` with lambdas

```
> let f x = x*2+1 in map f [1..5]
[3,5,7,9,11]
> map (\x -> x*2+1) [1..5]
[3,5,7,9,11]

> let p x = x `mod` 3 == 0
  in  filter p [1..10]
[3,6,9]
> filter (\x -> x `mod` 3 == 0) [1..10]
[3,6,9]
```

Operator sections

An operator section is an operator that has been partially applied:

`(+1)` is shorthand for `\x -> x+1`.

```
> :t (+1)
(+1) :: Num a => a -> a
> map (+1) [1..5]
[2,3,4,5,6]
> filter (>5) [1..10]
[6,7,8,9,10]
```

Three ways to write a program

```haskell
-- using list comprehension
result = [ f x | x <- xs , p x ]

-- using pattern matching + recursion
result = aux xs
  where
    aux [] = []
    aux (x:xs) | p x       = f x : aux xs
               | otherwise = aux xs

-- using higher-order functions
result = map f (filter p xs)
```

Live programming problem

Implement the following functions using higher-order functions:

- `applyFuns :: [a -> b] -> [a] -> [b]`
- `intersect :: Eq a => [a] -> [a] -> [a]`
- `allPairs :: [a] -> [b] -> [(a,b)]`

Higher-order functions: `all` and `any`

```
> :t all
all :: Foldable t =>
         (a -> Bool) -> t a -> Bool
> :t +d all
all :: (a -> Bool) -> [a] -> Bool
> :t +d any
any :: (a -> Bool) -> [a] -> Bool
> import Data.Char (isSpace)
> any isSpace "Hello, world!"
True
```

Higher-order functions: `takeWhile` and `dropWhile`

```
> :t takeWhile
takeWhile :: (a -> Bool) -> [a] -> [a]
> :t dropWhile
dropWhile :: (a -> Bool) -> [a] -> [a]
> dropWhile isSpace "  Hello, world!"
"Hello, world!"
```

Testing properties of functions

Higher-order properties

A higher-order property is a property that takes a **function** as an input:

```
prop_mapTwice
  :: (Int -> Int) -> [Int] -> Bool
prop_mapTwice f xs =
  map f (map f xs) == map (twice f) xs

-- prop> prop_mapTwice
-- No instance for (Show (Int -> Int))
```

QuickCheck tests properties by generating **random inputs**, but it cannot generate random functions.

Higher-order properties

A higher-order property is a property that takes a **function** as an input:

```
prop_mapTwice
  :: (Int -> Int) -> [Int] -> Bool
prop_mapTwice f xs =
  map f (map f xs) == map (twice f) xs

-- prop> prop_mapTwice
-- No instance for (Show (Int -> Int))
```

QuickCheck tests properties by generating **random inputs**, but it cannot generate random functions. *Or can it?*

Higher-order properties

QuickCheck provides the type `Fun a b` of shrinkable and printable functions.

```
prop_mapTwice
  :: Fun Int Int -> [Int] -> Bool
prop_mapTwice (Fn f) xs =
  map f (map f xs) == map (twice f) xs

-- prop> prop_mapTwice
-- +++ OK, passed 100 tests.
```

Generating functions with QuickCheck

```haskell
prop_bananas :: Fun String Int -> Bool
prop_bananas (Fn f) =
  f "banana" == f "monkey" ||
  f "banana" == f "elephant" ||
  f "monkey" == f "elephant"

-- prop> prop_bananas
-- *** Failed! Falsified
-- (after 5 tests and 163 shrinks):
-- {"banana"->2, "elephant"->0, _->1}
```

Identity and function composition

Function composition

```haskell
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Examples of how to use `(.)`:

```haskell
odd        = not . even
twice f    = f . f
sumsqeven  = sum
           . map (^2)
           . filter even
```

Building pipelines with `(.)`

We can use `(.)` to compose functions *without naming their arguments*:

```
processData = combine
            . map process
            . filter isValid
  where
    isValid = ...
    process = ...
    combine = ...
```

The identity function

Haskell's boringest function:

```
id :: a -> a
id x = x
```

One possible use case:

```
compose :: [a -> a] -> a -> a
compose []       = id
compose (f:fs) = f . compose fs
```

Question. How does Haskell compute `compose [(−3),(*2),(+5)] 3`?

The three laws of function composition

```haskell
id . f = f
```

```haskell
f . id = f
```

```haskell
f . (g . h) = (f . g) . h
```

This means Haskell functions form a category[2]

[2]See course CS4410 Category Theory for Programmers in CS master

The higher-order function `foldr`

A common pattern of recursion

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

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

or []             = False
or (b:bs)         = b || or bs

and []            = True
and (b:bs)        = b && and bs
```

Don't Repeat Yourself!

A common pattern of recursion

Many recursive functions on lists follow the following pattern:

```
f []     = v
f (x:xs) = x # f xs
```

The higher-order function `foldr` encapsulates this pattern. Instead of the above, we can simply write:

```
f = foldr (#) v
```

Examples of using `foldr`

```
sum     = foldr (+) 0

product = foldr (*) 1

or      = foldr (||) False

and     = foldr (&&) True
```

What does `foldr` do?

Intuition: `foldr (#) v` replaces each occurrence of `(:)` by `(#)` and the final `[]` by `v`:

```
  foldr (+) 0 [x1 ,  x2 ,  x3     ]
= foldr (+) 0 (x1 : (x2 : (x3 : [])))
=              x1 + (x2 + (x3 + 0 ))
```

Note that parentheses are associated to the right, hence the r in `foldr`.

Recursive Definition of `foldr`

```
foldr :: (a -> b -> b) -> b
      -> [a] -> b
foldr (#) v []     = v
foldr (#) v (x:xs) =
  x # (foldr (#) v xs)
```

Folding binary trees

For any recursive datatype we can define a folding function (not just for lists!)

```
foldt :: (a -> b) -> (b -> b -> b) -> 
         Tree a -> b
foldt w f (Leaf x)   = w x
foldt w f (Node l r) = f (foldt l) (foldt r)

occurs x = foldt (\y -> x == y) (||)
flatten  = foldt (\x -> [x])    (++)
```

`foldt w f` replaces each `Leaf` by `w` and each `Node` by `f`.

Folds in other languages

Folding functions exist in many other languages:

- Haskell: `foldr (+) 0 seq`
- Scala: `seq.fold(0)((a,b) => a + b)`
- Python: `reduce(lambda a,b:a+b, seq, 0)`
- Ruby: `seq.inject(0) {|a,b| a + b}`
- C#: `seq.Aggregate(func: (a,b) => a + b)`
- ...

More suspicious patterns (1/5)

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

3

More suspicious patterns (1/5)

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

```
length []     = 0
length (x:xs) =
  (\_ n -> 1 + n) x (length xs)
```

3

More suspicious patterns (1/5)

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

```
length []     = 0
length (x:xs) =
  (\_ n -> 1 + n) x (length xs)
```

```
length = foldr (\_ n -> 1+n) 0
```

Challenge. Implement `length` in point-free style (without using pattern matching, recursion, *or* lambdas).[3]

[3] The function `const` might be useful.

More suspicious patterns (2/5)

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

More suspicious patterns (2/5)

```
[]       ++ ys = ys
(x:xs)  ++ ys = x : (xs ++ ys)

[]       ++ ys = ys
(x:xs)  ++ ys = (:) x (xs ++ ys)
```

More suspicious patterns (2/5)

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
[]     ++ ys = ys
(x:xs) ++ ys = (:) x (xs ++ ys)
```

```
xs ++ ys = foldr (:) ys xs
```

Challenge. Implement `(++)` in point-free style.

More suspicious patterns (3/5)

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

More suspicious patterns (3/5)

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

```haskell
map f []     = []
map f (x:xs) =
  (\x ys -> f x : ys) x (map f xs)
```

More suspicious patterns (3/5)

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

```
map f []     = []
map f (x:xs) =
  (\x ys -> f x : ys) x (map f xs)
```

```
map f = foldr (\x ys -> f x : ys) []
```

Challenge. Implement `map` in point-free style.[4]

[4] **Warning:** pretty hard.

More suspicious patterns (4/5)

```haskell
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

More suspicious patterns (4/5)

```haskell
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```haskell
reverse []     = []
reverse (x:xs) =
  (\x xs -> xs ++ [x]) x (reverse xs)
```

More suspicious patterns (4/5)

```haskell
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```haskell
reverse []     = []
reverse (x:xs) =
  (\x xs -> xs ++ [x]) x (reverse xs)
```

```haskell
reverse = foldr (\x xs -> xs ++ [x]) []
```

Challenge. Implement `reverse` in point-free style.[5]

[5] **Warning:** very hard, you might need the function `pure`.

More suspicious patterns (5/5)

```
filter p []     = []
filter p (x:xs) =
    if   p x
    then x : filter p xs
    else filter p xs
```

More suspicious patterns (5/5)

```haskell
filter p []     = []
filter p (x:xs) =
    if   p x
    then x : filter p xs
    else filter p xs

filter p = foldr (\x xs' -> if p x
                            then x : xs'
                            else xs') []
```

Challenge. Implement `filter` in point-free style.[6]

[6] **Warning:** *really* hard, you might need `bool` and `(&&&)`.

Quiz question

Question. How can we implement `concat :: [[a]] -> [a]` using `foldr`?

1. `concat = foldr (:) []`
2. `concat = foldr (++) []`
3. `concat = foldr (:) [[]]`
4. `concat = foldr (++) [[]]`

Extra benefit of `foldr`: optimizations

Some advanced compiler optimizations are easier on programs with `foldr`:

- `foldr` fusion:

```
foldr f v (map g xs)
 == foldr (\x y -> f (g x) y) xs
```

- The 'banana split rule':[7]

```
(sum xs, length xs)
 == foldr (\n (x,y) -> (n+x,1+y))
                (0,0)
```

[7] E. Meijer et. al. (1991): *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*

From the book: Binary string transmitter

Goal: Simulate transmission of a string of characters encoded as a list of binary numbers.

```
bin2int :: [Int] -> Int
bin2int = foldr (\x y -> x + 2*y)

int2bin :: Int -> [Int]
int2bin 0 = []
int2bin n =
  n `mod` 2 : int2bin (n `div` 2)
```

See section 7.6 of the book for the full example.

A discussion on `foldr`

Suppose a fellow student makes the following claim:

> *"All recursive functions on lists can be written in terms of `foldr`."*

Question. Do you agree or disagree? Why?

The `foldl` function

`foldl` is a version of `foldr` that associates to the left:

```
foldl (+) 0 [x1,x2,x3]
== ((0 + x1) + x2) + x3
```

The difference is significant for non-associative operations such as `(-)`:

```
foldr (-) 0 [1,2,3] = 1-(2-(3-0)) = 2
foldl (-) 0 [1,2,3] = ((0-1)-2)-3 = -6
```

Live programming exercise

Exercise. Implement the function `reverse :: [a] -> [a]` using `foldl`

Recursive definition of `foldl`

```
foldl :: (b -> a -> b) -> b
           -> [a] -> b
foldl (#) v []       = v
foldl (#) v (x:xs) =
  foldl (#) (v # x) xs
```

Question: Can we define `foldl f = foldr (flip f)`? If not, can we define `foldl` in terms of `foldr` in some other way?

The problem with `foldl`

Warning: The `foldl` function is notorious for causing performance problems, in particular transient space leaks.

The cause for this is Haskell's lazy evaluation strategy (see week 5).

The prelude provides a strict version `foldl'` (with the same type as `fold`) that is almost always more efficient.

What's next?

Next lecture: Type classes

To do:

- Read the book:
    - Today: sections 4.5-4.6, 7.1-7.5
    - Next lecture: 8.5, 12.1-12.2
- Read the binary string transmitter example in section 7.6 of the book
- Continue on week 2 exercises on Weblab