[CSE301 / Lecture 2]
Higher-order functions and type classes

Noam Zeilberger

Ecole Polytechnique

14 September 2022

What are is a higher-order function?

A function that takes one or more functions as input.

Main motivation: expressing the **common denominator** between a collection of first-order functions, thus promoting code reuse!

Learning tip: HO functions may be hard to grasp at first, but will eventually help in “seeing the forest for the trees”.

First example: abstracting case-analysis

Recall that given $f :: a \to c$ and $g :: b \to c$, we can define

$$
\begin{array}{l}
h :: \mathit{Either}\ a\ b \to c \\
h\ (\mathit{Left}\ x) = f\ x \\
h\ (\mathit{Right}\ y) = g\ y
\end{array}
$$

In other words, can define h by case-analysis.

For example:

$$
\begin{array}{l}
\mathit{asInt} :: \mathit{Either}\ \mathit{Bool}\ \mathit{Int} \to \mathit{Int} \\
\mathit{asInt}\ (\mathit{Left}\ b) = \mathbf{if}\ b\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\
\mathit{asInt}\ (\mathit{Right}\ n) = n \\
\mathit{isBool} :: \mathit{Either}\ \mathit{Bool}\ \mathit{Int} \to \mathit{Bool} \\
\mathit{isBool}\ (\mathit{Left}\ b) = \mathit{True} \\
\mathit{isBool}\ (\mathit{Right}\ n) = \mathit{False}
\end{array}
$$

First example: abstracting case-analysis

The Prelude defines a higher-order function that "internalizes" the principle of case-analysis over sum types, so to speak:

$$\begin{aligned}
&\mathit{either} :: (a \to c) \to (b \to c) \to \mathit{Either}\ a\ b \to c \\
&\mathit{either}\ f\ g\ (\mathit{Left}\ x) = f\ x \\
&\mathit{either}\ f\ g\ (\mathit{Right}\ y) = g\ y
\end{aligned}$$

Here is how we can redefine *asInt* and *isBool* using *either* (and λ):

$$\begin{aligned}
&\mathit{asInt} = \mathit{either}\ (\backslash b \to \mathbf{if}\ b\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ (\backslash n \to n) \\
&\mathit{isBool} = \mathit{either}\ (\backslash b \to \mathit{True})\ (\backslash n \to \mathit{False})
\end{aligned}$$

Whereas before we could spot that the two functions were instances of a simple common "design pattern", now they are literally two applications of the same (higher-order) function.

First example: abstracting case-analysis

Here again:

$$\begin{array}{l} \mathit{asInt} = \mathit{either}\ (\backslash b \rightarrow \textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0)\ (\backslash n \rightarrow n) \\ \mathit{isBool} = \mathit{either}\ (\backslash b \rightarrow \mathit{True})\ (\backslash n \rightarrow \mathit{False}) \end{array}$$

Observe we only partially applied *either*. Alternatively:

$$\begin{array}{l} \mathit{asInt}\ v = \mathit{either}\ (\backslash b \rightarrow \textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0)\ (\backslash n \rightarrow n)\ v \\ \mathit{isBool}\ v = \mathit{either}\ (\backslash b \rightarrow \mathit{True})\ (\backslash n \rightarrow \mathit{False})\ v \end{array}$$

but these two versions are completely equivalent.

(They are said to be "η-equivalent".)

First example: abstracting case-analysis

Finally, recall arrow associates to the right by default:

$$either :: (a \to c) \to ((b \to c) \to (Either\ a\ b \to c))$$

The type of *either* looks a lot like

$$(A \supset C) \supset ([B \supset C] \supset [(A \vee B) \supset C])$$

which you can verify is a tautology. (This is a recurring theme!)

Second example: mapping over a list

Consider the following first-order functions on lists...

Second example: mapping over a list

(Add one to every element in a list of integers.)

$$\begin{aligned}&\mathit{mapAddOne} :: [\mathit{Integer}] \to [\mathit{Integer}]\\&\mathit{mapAddOne}\ [\,] = [\,]\\&\mathit{mapAddOne}\ (x : \mathit{xs}) = (1 + x) : \mathit{mapAddOne}\ \mathit{xs}\end{aligned}$$

Example: $\mathit{mapAddOne}\ [1\mathinner{\ldotp\ldotp}5] = [2, 3, 4, 5, 6]$

Second example: mapping over a list

(Square every element in a list of integers.)

```
mapSquare :: [Integer] → [Integer]
mapSquare [] = []
mapSquare (x : xs) = (x ∗ x) : mapSquare xs
```

Example: *mapSquare* $[1\,..\,5] = [1, 4, 9, 16, 25]$

Second example: mapping over a list

(Compute the length of each list in a list of lists.)

$$
\begin{array}{l}
\mathit{mapLength} :: [[a]] \rightarrow [\mathit{Int}] \\
\mathit{mapLength}\ [\,] = [\,] \\
\mathit{mapLength}\ (x : \mathit{xs}) = \mathit{length}\ x : \mathit{mapLength}\ \mathit{xs}
\end{array}
$$

Example: $\mathit{mapLength}\ [\texttt{"hello"}, \texttt{"world!"}] = [5, 6]$‘

Second example: mapping over a list

GCD = "apply some transformation to every element of a list"

We can internalize this as a higher-order function:

$$map :: (a \to b) \to [a] \to [b]$$
$$map\ f\ [\,] = [\,]$$
$$map\ f\ (x : xs) = (f\ x) : map\ f\ xs$$

For example:

$$mapAddOne = map\ (1+)$$
$$mapSquare = map\ (\backslash n \to n * n)$$
$$mapLength = map\ length$$

Some useful functions on functions

The "currying" and "uncurrying" principles:

$$\begin{aligned}
&\mathit{curry} :: ((a, b) \to c) \to (a \to b \to c) \\
&\mathit{curry}\ f\ x\ y = f\ (x, y) \\
&\mathit{uncurry} :: (a \to b \to c) \to ((a, b) \to c) \\
&\mathit{uncurry}\ g\ (x, y) = g\ x\ y
\end{aligned}$$

Or equivalently:

$$\begin{aligned}
&\mathit{curry}\ f = \backslash x \to \backslash y \to f\ (x, y) \\
&\mathit{uncurry}\ g = \backslash (x, y) \to g\ x\ y
\end{aligned}$$

Example: $\mathit{map}\ (\mathit{uncurry}\ (+))\ [(0, 1), (2, 3), (4, 5)] = [1, 5, 9]$

Logically: $(A \wedge B) \supset C \iff A \supset (B \supset C)$.

Some useful functions on functions

The principle of sequential composition:

$$(\circ) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$(g \circ f)\ x = g\ (f\ x)$$

Example: $map\ ((+1) \circ (*2))\ [0 \mathinner{..} 4] = [1, 3, 5, 7, 9]$

Logically: transitivity of implication.

Some useful functions on functions

The principle of exchange:

$$\begin{aligned}&\mathit{flip} :: (a \to b \to c) \to (b \to a \to c)\\ &\mathit{flip}\ f\ x\ y = f\ y\ x\end{aligned}$$

The principle of weakening:

$$\begin{aligned}&\mathit{const} :: b \to (a \to b)\\ &\mathit{const}\ x\ y = x\end{aligned}$$

The principle of contraction:

$$\begin{aligned}&\mathit{dupl} :: (a \to a \to b) \to (a \to b)\\ &\mathit{dupl}\ f\ x = f\ x\ x\end{aligned}$$

More higher-order functions on lists

The Haskell Prelude and Standard Library define a number of HO functions that capture common ways of manipulating lists...

More higher-order functions on lists

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

Examples:

```
> filter (>3) [1..5]
[4, 5]
> filter Data.Char.isUpper "Glasgow Haskell Compiler"
"GHC"
```

More higher-order functions on lists

```
all, any :: (a → Bool) → [a] → Bool
all p []  = True
all p (x : xs) = p x && all p xs
any p [] = False
any p (x : xs) = p x || any p xs
```

Examples: *all* $(>3)\ [1\mathinner{..}5] =$ *False*, *any* $(>3)\ [1\mathinner{..}5] =$ *True*.

More higher-order functions on lists

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

Examples: *takeWhile* $(>3)\ [1\mathinner{..}5] = [\,]$,
takeWhile $(<3)\ [1\mathinner{..}5] = [1, 2]$,
dropWhile $(<3)\ [1\mathinner{..}5] = [3, 4, 5]$.

More higher-order functions on lists

```
concatMap :: (a → [b]) → [a] → [b]
concatMap f [ ] = [ ]
concatMap f (x : xs) = f x ++ concatMap f xs
```

Examples:

```
> concatMap (\x → [x]) [1 . . 5]
[1, 2, 3, 4, 5]
> concatMap (\x → if x `mod` 2 ≡ 1 then [x] else [ ]) [1 . . 5]
[1, 3, 5]
> concatMap (\x → concatMap (\y → [x . . y]) [1 . . 3]) [1 . . 3]
[1, 1, 2, 1, 2, 3, 2, 2, 3, 3]
```

Note $\mathit{concatMap}\ f = \mathit{concat} \circ \mathit{map}\ f$.

foldr: the Swiss army knife of list functions

Remarkably, all of the preceding higher-order list functions, and many other functions besides, can be defined as instances of a single higher-order function!

foldr: the Swiss army knife of list functions

Suppose want to write a function $[a] \to b$ *inductively* over lists.

We provide a "base case" $v :: b$.

We provide an "inductive step" $f :: a \to b \to b$.

Putting these together, we get a recursive definition:

$$\begin{array}{l} h :: [a] \to b \\ h\ [\,] = v \\ h\ (x :: xs) = f\ x\ (h\ xs) \end{array}$$

foldr: the Swiss army knife of list functions

Since this schema is completely generic in the “base case” and the “inductive step”, we can internalize it as a higher-order function:

$$
\begin{array}{l}
\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
\mathit{foldr}\ f\ v\ [\,] = v \\
\mathit{foldr}\ f\ v\ (x : xs) = f\ x\ (\mathit{foldr}\ f\ v\ xs)
\end{array}
$$

Here are some examples:

$$
\begin{array}{l}
\mathit{filter}\ p = \mathit{foldr}\ (\backslash x\ xs \to \mathbf{if}\ p\ x\ \mathbf{then}\ x : xs\ \mathbf{else}\ xs)\ [\,] \\
\mathit{all}\ p = \mathit{foldr}\ (\backslash x\ b \to p\ x\ \&\&\ b)\ \mathit{True} \\
\mathit{takeWhile}\ p = \mathit{foldr}\ (\backslash x\ xs \to \mathbf{if}\ p\ x\ \mathbf{then}\ x : xs\ \mathbf{else}\ [\,])\ [\,] \\
\mathit{concatMap}\ f = \mathit{foldr}\ (\backslash x\ ys \to f\ x +\!\!+\ ys)\ [\,]
\end{array}
$$

And let’s look at some more...

foldr: the Swiss army knife of list functions

$$
\begin{aligned}
&\mathit{sum} :: \mathit{Num}\ a \Rightarrow [a] \rightarrow a\\
&\mathit{sum}\ [\,] = 0\\
&\mathit{sum}\ (x : \mathit{xs}) = x + \mathit{sum}\ \mathit{xs}
\end{aligned}
$$

may be summarized as:

$$\mathit{sum} = \mathit{foldr}\ (+)\ 0$$

foldr: the Swiss army knife of list functions

$$\begin{aligned}&\mathit{product} :: \mathit{Num}\ a \Rightarrow [a] \rightarrow a\\&\mathit{product}\ [\,] = 1\\&\mathit{product}\ (x : \mathit{xs}) = x * \mathit{product}\ \mathit{xs}\end{aligned}$$

may be summarized as:

$$\mathit{product} = \mathit{foldr}\ (*)\ 1$$

foldr: the Swiss army knife of list functions

$$\begin{aligned}
&\mathit{length} :: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] = 0 \\
&\mathit{length}\ (x : \mathit{xs}) = 1 + \mathit{length}\ \mathit{xs}
\end{aligned}$$

may be summarized as:

$$\mathit{length} = \mathit{foldr}\ (\backslash x\ n \to 1 + n)\ 0 = \mathit{foldr}\ (\mathit{const}\ (1+))\ 0$$

foldr: the Swiss army knife of list functions

$$
\begin{aligned}
&\mathit{concat} :: [[a]] \to [a] \\
&\mathit{concat}\ [\,] = [\,] \\
&\mathit{concat}\ (\mathit{xs} : \mathit{xss}) = \mathit{xs} +\!\!+ \mathit{concat}\ \mathit{xss}
\end{aligned}
$$

may be summarized as:

$$\mathit{concat} = \mathit{foldr}\ (+\!\!+)\ [\,]$$

foldr: the Swiss army knife of list functions

$$
\begin{aligned}
&\mathit{copy} :: [a] \rightarrow [a] \\
&\mathit{copy}\ [\,] = [\,] \\
&\mathit{copy}\ (x : \mathit{xs}) = x : \mathit{copy}\ \mathit{xs}
\end{aligned}
$$

may be summarized as:

$$\mathit{copy} = \mathit{foldr}\ (:)\ [\,]$$

foldr: the Swiss army knife of list functions

(a somewhat more subtle example:)

$$
\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a] \\
&[\,] +\!\!+ \mathit{ys} = \mathit{ys} \\
&(x : \mathit{xs}) +\!\!+ \mathit{ys} = x : (\mathit{xs} +\!\!+ \mathit{ys})
\end{aligned}
$$

may be summarized as:

$$(+\!\!+) = \mathit{foldr}\ (\backslash x\ g \to (x{:}) \circ g)\ \mathit{id}$$

Aside: folding from the left

$$\begin{aligned}
&\mathit{foldr}\ (+)\ 0\ [1,2,3,4,5] \\
&= 1 + \mathit{foldr}\ (+)\ 0\ [2,3,4,5] \\
&= 1 + (2 + \mathit{foldr}\ (+)\ 0\ [3,4,5] \\
&= 1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [4,5] \\
&= 1 + (2 + (3 + (4 + \mathit{foldr}\ (+)\ 0\ [5] \\
&= 1 + (2 + (3 + (4 + (5 + \mathit{foldr}\ (+)\ 0\ [\,]))))) \\
&= 1 + (2 + (3 + (4 + (5 + 0)))) \\
&= 1 + (2 + (3 + (4 + 5))) \\
&= 1 + (2 + (3 + 9)) \\
&= 1 + (2 + 12) \\
&= 1 + 14 \\
&= 15
\end{aligned}$$

Observe that additions are performed right-to-left.

Aside: folding from the left

Sometimes we want to go left-to-right:

$$\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to [a] \to b \\
&\mathit{foldl}\ f\ v\ [\,] = v \\
&\mathit{foldl}\ f\ v\ (x : xs) = \mathit{foldl}\ f\ (f\ v\ x)\ xs
\end{aligned}$$

Example:

$$\begin{aligned}
&\mathit{foldl}\ (+)\ 0\ [1,2,3,4,5] \\
&= \mathit{foldl}\ (+)\ 1\ [2,3,4,5] \\
&= \mathit{foldl}\ (+)\ 3\ [3,4,5] \\
&= \mathit{foldl}\ (+)\ 6\ [4,5] \\
&= \mathit{foldl}\ (+)\ 10\ [5] \\
&= \mathit{foldl}\ (+)\ 15\ [\,] \\
&= 15
\end{aligned}$$

(Q: does this remind you of something from Lecture 1?)

Higher-order functions over trees

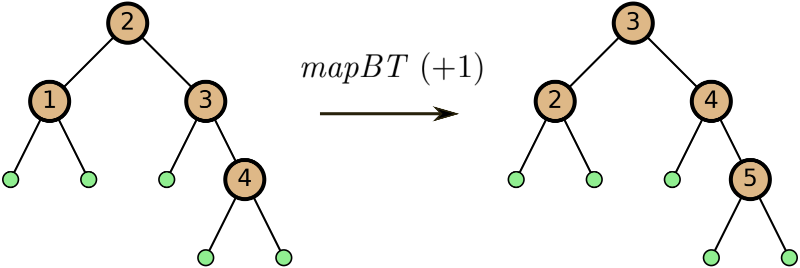
Recall our data type of binary trees with labelled nodes:

```
data BinTree a = Leaf | Node a (BinTree a) (BinTree a)
  deriving (Show, Eq)
```

It supports a natural analogue of the *map* function on lists:

```
mapBT :: (a → b) → BinTree a → BinTree b
mapBT f Leaf = Leaf
mapBT f (Node x tL tR) = Node (f x)
  (mapBT f tL) (mapBT f tR)
```

Higher-order functions over trees



Higher-order functions over trees

It also supports a natural analogue of *foldr*:

$$
\begin{aligned}
&\mathit{foldBT} :: b \to (a \to b \to b \to b) \to \mathit{BinTree}\ a \to b \\
&\mathit{foldBT}\ v\ f\ \mathit{Leaf} = v \\
&\mathit{foldBT}\ v\ f\ (\mathit{Node}\ x\ tL\ tR) = f\ x \\
&\quad (\mathit{foldBT}\ v\ f\ tL)\ (\mathit{foldBT}\ v\ f\ tR)
\end{aligned}
$$

For example:

$$
\begin{aligned}
&\mathit{nodes} = \mathit{foldBT}\ 0\ (\backslash x\ m\ n \to 1 + m + n) \\
&\mathit{leaves} = \mathit{foldBT}\ 1\ (\backslash x\ m\ n \to m + n) \\
&\mathit{height} = \mathit{foldBT}\ 0\ (\backslash x\ m\ n \to 1 + \mathit{max}\ m\ n) \\
&\mathit{mirror} = \mathit{foldBT}\ \mathit{Leaf}\ (\backslash x\ tL'\ tR' \to \mathit{Node}\ x\ tR'\ tL')
\end{aligned}
$$

Type classes

By now we've seen several examples of polymorphic functions with type class constraints, e.g.:

$$
\begin{aligned}
&\mathit{sort} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a] \\
&\mathit{lookup} :: \mathit{Eq}\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow \mathit{Maybe}\ b \\
&\mathit{sum}, \mathit{product} :: \mathit{Num}\ a \Rightarrow [a] \rightarrow a
\end{aligned}
$$

Intuitively, these constraints express minimal requirements on the type a needed to define these functions.

Type classes

Formally, a type class is defined by specifying the type signatures of operations, possibly together with default implementations of some operations in terms of others. For example:

```
class Eq a where
  (≡), (≢) :: a → a → Bool
  x ≢ y = not (x ≡ y)
  x ≡ y = not (x ≢ y)
```

We show the constraint is satisfied by providing an instance, e.g.:

```
instance Eq Bool where
  False ≡ b = not b
  True ≡ b = b
```

Type classes

Sometimes need hereditary constraints to define instances:

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

Type classes

Possible for one class to inherit from another, e.g.:

```
class Eq a ⇒ Ord a where
  compare :: a → a → Ordering
  (<), (⩽), (>), (⩾) :: a → a → Bool
  max, min :: a → a → a

  compare x y = if x ≡ y then EQ
    else if x ⩽ y then LT
    else GT

  x < y = case compare x y of { LT → True; _ → False }
  x ⩽ y = case compare x y of { GT → False; _ → True }
  x > y = case compare x y of { GT → True; _ → False }
  x ⩾ y = case compare x y of { LT → False; _ → True }

  max x y = if x ⩽ y then y else x
  min x y = if x ⩽ y then x else y
```

Type classes

(That looks complicated, but basically you only need to implement ($\leqslant$) to define an *Ord* instance, assuming you already have *Eq*.)

Type classes

Often implicit that operations should obey certain laws.

For example, $(\equiv)$ should be reflexive, symmetric, and transitive.

Similarly, $(\leqslant)$ should be a total ordering.

These expectations may be described in the documentation of a type class, but are not enforced by the language.

Type classes

Type classes are a very cool feature of Haskell, but it is worth mentioning that in a certain sense they may be seen as “just” a convenient mechanism for defining higher-order functions.

Indeed, a constraint may always be replaced by the types of the operations in (a minimal defn of) the corresponding type class...

Type classes

Replace $sort :: Ord\ a \Rightarrow [a] \rightarrow [a]$ by

$$sortHO :: (a \rightarrow a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$$

Replace $lookup :: Eq\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow Maybe\ b$ by

$$lookupHO :: (a \rightarrow a \rightarrow Bool) \rightarrow a \rightarrow [(a, b)] \rightarrow Maybe\ b$$

etc.

Type classes

Though this translation does have certain drawbacks...

Most significantly, every call to a function with constraints would have to pass extra arguments, corresponding to the implementation of the type class instance.

In effect, type classes are useful because these “semantically implicit” arguments are automatically inferred by the type checker.

(Although that too has its drawbacks – for example in Haskell it is only possible to define *one* instance of a type class for a given type, although there may not be a single canonical instance.)