[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

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

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

For example:

$$\begin{aligned}
&asInt :: Either\ Bool\ Int \to Int\\
&asInt\ (Left\ b) = \mathbf{if}\ b\ \mathbf{then}\ 1\ \mathbf{else}\ 0\\
&asInt\ (Right\ n) = n
\end{aligned}$$

$$\begin{aligned}
&isBool :: Either\ Bool\ Int \to Bool\\
&isBool\ (Left\ b) = True\\
&isBool\ (Right\ n) = False
\end{aligned}$$

First example: abstracting case-analysis

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

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

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

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

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:

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

Observe we only partially applied *either*. Alternatively:

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

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

Second example: mapping over a list

(Square every element in a list of integers.)

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

Example: $\mathit{mapSquare}\ [1 \mathinner{..} 5] = [1, 4, 9, 16, 25]$

Second example: mapping over a list

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

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

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

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:

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

For example:

$$
\begin{aligned}
&\mathit{mapAddOne} = \mathit{map}\ (1+) \\
&\mathit{mapSquare} = \mathit{map}\ (\backslash n \to n * n) \\
&\mathit{mapLength} = \mathit{map}\ \mathit{length}
\end{aligned}
$$

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

Logically: transitivity of implication.

Some useful functions on functions

The principle of exchange:

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

The principle of weakening:

$$\mathit{const} :: b \to (a \to b)$$
$$\mathit{const}\ x\ y = x$$

The principle of contraction:

$$\mathit{dupl} :: (a \to a \to b) \to (a \to b)$$
$$\mathit{dupl}\ f\ x = f\ x\ x$$

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

$$
\begin{aligned}
&\mathit{filter} :: (a \to \mathit{Bool}) \to [a] \to [a] \\
&\mathit{filter}\ p\ [\,] = [\,] \\
&\mathit{filter}\ p\ (x : \mathit{xs}) \\
&\quad \mid p\ x = x : \mathit{filter}\ p\ \mathit{xs} \\
&\quad \mid \mathit{otherwise} = \mathit{filter}\ p\ \mathit{xs}
\end{aligned}
$$

Examples:

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

More higher-order functions on lists

$$
\begin{aligned}
&\mathit{all}, \mathit{any} :: (a \to \mathit{Bool}) \to [a] \to \mathit{Bool} \\
&\mathit{all}\ p\ [\,] \ \ = \mathit{True} \\
&\mathit{all}\ p\ (x : xs) = p\ x\ \&\&\ \mathit{all}\ p\ xs \\
&\mathit{any}\ p\ [\,] = \mathit{False} \\
&\mathit{any}\ p\ (x : xs) = p\ x \mathbin{||} \mathit{any}\ p\ xs
\end{aligned}
$$

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

More higher-order functions on lists

$$
\begin{aligned}
& takeWhile, dropWhile :: (a \to Bool) \to [a] \to [a] \\
& takeWhile\ p\ [\,] = [\,] \\
& takeWhile\ p\ (x : xs) \\
& \quad \mid p\ x \qquad\quad = x : takeWhile\ p\ xs \\
& \quad \mid otherwise = \ [\,] \\
& dropWhile\ p\ [\,] = [\,] \\
& dropWhile\ p\ (x : xs) \\
& \quad \mid p\ x \qquad\quad = dropWhile\ p\ xs \\
& \quad \mid otherwise = \ x : xs
\end{aligned}
$$

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

More higher-order functions on lists

$$
\begin{aligned}
&\mathit{concatMap} :: (a \to [b]) \to [a] \to [b] \\
&\mathit{concatMap}\ f\ [\,] = [\,] \\
&\mathit{concatMap}\ f\ (x : xs) = f\ x + \mkern-10mu + \ \mathit{concatMap}\ f\ xs
\end{aligned}
$$

Examples:

$$
\begin{aligned}
&> \mathit{concatMap}\ (\backslash x \to [x])\ [1\mathinner{\ldotp\ldotp}5] \\
&[1,2,3,4,5] \\
&> \mathit{concatMap}\ (\backslash x \to \textbf{if}\ x\ `\mathit{mod}`\ 2 \equiv 1\ \textbf{then}\ [x]\ \textbf{else}\ [\,])\ [1\mathinner{\ldotp\ldotp}5] \\
&[1,3,5] \\
&> \mathit{concatMap}\ (\backslash x \to \mathit{concatMap}\ (\backslash y \to [x\mathinner{\ldotp\ldotp}y])\ [1\mathinner{\ldotp\ldotp}3])\ [1\mathinner{\ldotp\ldotp}3] \\
&[1,1,2,1,2,3,2,2,3,3]
\end{aligned}
$$

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

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:

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

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:

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

Here are some examples:

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

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

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

may be summarized as:

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

foldr: the Swiss army knife of list functions

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

may be summarized as:

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

foldr: the Swiss army knife of list functions

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

may be summarized as:

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

foldr: the Swiss army knife of list functions

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

may be summarized as:

$$copy = foldr\ (:)\ [\,]$$

foldr: the Swiss army knife of list functions

(a somewhat more subtle example:)

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

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

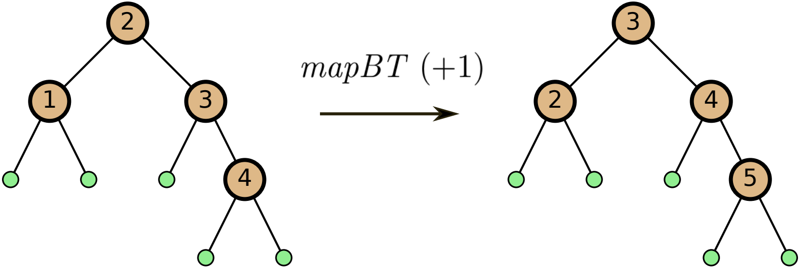
Recall our data type of binary trees with labelled nodes:

$$\textbf{data}\ \mathit{BinTree}\ a = \mathit{Leaf} \mid \mathit{Node}\ a\ (\mathit{BinTree}\ a)\ (\mathit{BinTree}\ a)$$
$$\quad \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})$$

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

$$\mathit{mapBT} :: (a \to b) \to \mathit{BinTree}\ a \to \mathit{BinTree}\ b$$
$$\mathit{mapBT}\ f\ \mathit{Leaf} = \mathit{Leaf}$$
$$\mathit{mapBT}\ f\ (\mathit{Node}\ x\ tL\ tR) = \mathit{Node}\ (f\ x)$$
$$\quad (\mathit{mapBT}\ f\ tL)\ (\mathit{mapBT}\ f\ tR)$$

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: what are they?

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

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

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

Type classes: what are they?

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:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ a\ \textbf{where} \\
\quad (\equiv), (\not\equiv) :: a \to a \to \mathit{Bool} \\
\quad x \not\equiv y = \mathit{not}\ (x \equiv y) \\
\quad x \equiv y = \mathit{not}\ (x \not\equiv y)
\end{array}
$$

Type class instances

We show the constraint is satisfied by providing an *instance:*

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ \mathit{Bool}\ \textbf{where} \\
\quad x \equiv y = \textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ \mathit{not}\ y
\end{array}
$$

Sometimes need hereditary constraints to define instances:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]\ \textbf{where} \\
\quad [\,] \equiv [\,] = \mathit{True} \\
\quad (x : xs) \equiv (y : ys) = x \equiv y \mathbin{\&\&} xs \equiv ys \\
\quad \_ \equiv \_ = \mathit{False}
\end{array}
$$

Class hierarchy

Possible for one type class to inherit from another, e.g.:[1]

$$
\begin{aligned}
&\textbf{class } \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a \textbf{ where}\\
&\quad \mathit{compare} :: a \to a \to \mathit{Ordering}\\
&\quad (<), (\leqslant), (>), (\geqslant) :: a \to a \to \mathit{Bool}\\
&\quad \mathit{max}, \mathit{min} :: a \to a \to a\\
&\quad \mathit{compare}\ x\ y = \textbf{if } x \equiv y \textbf{ then } \mathit{EQ}\\
&\qquad \textbf{else if } x \leqslant y \textbf{ then } \mathit{LT}\\
&\qquad \textbf{else } \mathit{GT}\\
&\quad x < y = \textbf{case } \mathit{compare}\ x\ y \textbf{ of } \{\mathit{LT} \to \mathit{True};\ \_ \to \mathit{False}\}\\
&\quad x \leqslant y = \textbf{case } \mathit{compare}\ x\ y \textbf{ of } \{\mathit{GT} \to \mathit{False};\ \_ \to \mathit{True}\}\\
&\quad x > y = \textbf{case } \mathit{compare}\ x\ y \textbf{ of } \{\mathit{GT} \to \mathit{True};\ \_ \to \mathit{False}\}\\
&\quad x \geqslant y = \textbf{case } \mathit{compare}\ x\ y \textbf{ of } \{\mathit{LT} \to \mathit{False};\ \_ \to \mathit{True}\}\\
&\quad \mathit{max}\ x\ y = \textbf{if } x \leqslant y \textbf{ then } y \textbf{ else } x\\
&\quad \mathit{min}\ x\ y = \textbf{if } x \leqslant y \textbf{ then } x \textbf{ else } y
\end{aligned}
$$

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

Laws

It is 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 Haskell language.[2]

[2] Although they can be enforced in dependently typed languages!

Type classes from higher-order functions

Type classes are a cool feature of Haskell, but in a certain sense they may be seen as “just” a convenient mechanism for defining higher-order functions, since a constraint may always be replaced by the types of the operations in (a minimal definition of) the corresponding type class...

Type classes from higher-order functions

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

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

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

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

and so on.

Whenever we would call a function with constraints, we instead call a HO function while providing one or more extra arguments.

Automatic type class resolution

Drawback of this translation: every call to a function with constraints has to pass potentially many extra arguments!

Type classes are useful because these "semantically implicit" arguments are automatically inferred by the type checker.

> $>$ **import** *Data.List*
> $>$ *sort* $[3, 1, 4, 1, 5, 9]$
> $[1, 1, 3, 4, 5, 9]$
> $>$ *sort* ["my", "dog", "has", "fleas"]
> ["dog", "fleas", "has", "my"]

Unfortunately, it is only possible to define a single instance of a type class for a given type, although we can get around this with the **newtype** mechanism...

Behaves similarly to a **data** definition but only allowed to have a single constructor with a single argument. The purpose is to introduce an *isomorphic copy* of another type.

$$\begin{array}{l}
\textbf{newtype}\ \mathit{Sum}\ a = \mathit{Sum}\ a \\
\textbf{newtype}\ \mathit{Product}\ a = \mathit{Product}\ a \\
\textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Monoid}\ (\mathit{Sum}\ a)\ \textbf{where} \\
\quad \mathit{mempty} = \mathit{Sum}\ 0 \\
\quad \mathit{mappend}\ (\mathit{Sum}\ x)\ (\mathit{Sum}\ y) = \mathit{Sum}\ (x + y) \\
\textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Monoid}\ (\mathit{Product}\ a)\ \textbf{where} \\
\quad \mathit{mempty} = \mathit{Product}\ 1 \\
\quad \mathit{mappend}\ (\mathit{Product}\ x)\ (\mathit{Product}\ y) = \mathit{Product}\ (x * y)
\end{array}$$