CSE301(Functional Programming)

Lecture 2: Higher-order functions and type classes

Noam Zeilberger

version: September 15, 2024

A **higher-order function** is a function that takes one or more functions as input. A primary motivation for defining a higher-order function is to express the common denominator between a collection of first-order functions. Higher-order functional programming is thus a way to promote code reuse. Although it may take you a bit more time to understand what a higher-order function is doing than the corresponding first-order ones, eventually it will help you in "seeing the forest for the trees", and hopefully give you insights into solving certain kinds of programming problems much more quickly.

1 0th example: applying a function twice

We can write a simple higher-order function that takes a function and applies it twice to some argument.

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

For example,[1] we have

$$\begin{aligned} \mathit{twice}\ (1+)\ 3 &= 5 \\ \mathit{twice}\ (\backslash n \to n * n)\ 3 &= 81 \\ \mathit{twice}\ \mathit{reverse}\ [1,2,3] &= [1,2,3] \end{aligned}$$

Now let's consider some more interesting examples.

2 First example: abstracting case-analysis

As we discussed in Lecture 1, sum types obey the general principle that if $f :: a \to c$ and $g :: b \to c$ are two functions taking types a and b to the same target type c, then there is a canonical function

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

[1] In Haskell, the expression $(1+)$ is called a (left) *section* of the infix operator $(+)$. It is equivalent to the lambda expression $\backslash n \to 1 + n$. The lambda expression $\backslash n \to n * n$ denotes the function which squares its argument. We will talk more about lambda notation in Lecture 3.

taking the sum type to the same target, defined by pattern-matching on the *Left* or *Right* constructor (or in other words, by case-analysis). Here again are two concrete examples of functions on the sum type *Either Bool Int*:

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

The Haskell Prelude defines a higher-order function that "internalizes" the principle of case-analysis over sum types by taking the functions f and g as extra arguments:

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

Here is how we can redefine *asInt* and *isBool* using *either* and lambda notation:

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

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.

Observe that here we have only partially applied the function *either* to a pair of functions f and g, to compute new functions *either* f g. Alternatively, we could have defined

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

supplying an extra argument $v :: \mathit{Either\ Bool\ Int}$, but in Haskell these two versions are completely equivalent. (They are said to be *η-equivalent*, in lambda calculus jargon.) Finally, recall that the arrow type constructor associates to the right by default, so that the type of *either* may be equivalently written with extra parentheses as:

$$\mathit{either} :: (a \to c) \to ((b \to c) \to (\mathit{Either}\ a\ b \to c))$$

In this form it looks a lot like the logical formula

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

which you can easily verify is a tautology. And indeed this analogy can be made precise, with many more examples of higher-order functions corresponding to proofs of tautologies as we will see shortly.

3 Second example: mapping over a list

Consider the following first-order functions on lists:

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

$$mapAddOne :: [Integer] \to [Integer]$$

$$mapAddOne\ [\,] = [\,]$$
$$mapAddOne\ (x : xs) = (1 + x) : mapAddOne\ xs$$

Example: $mapAddOne\ [1\,..\,5] = [2, 3, 4, 5, 6]$

(Square every element in a list of integers.)

$$mapSquare :: [Integer] \to [Integer]$$

$$mapSquare\ [\,] = [\,]$$
$$mapSquare\ (x : xs) = (x * x) : mapSquare\ xs$$

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

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

$$mapLength :: [[a]] \to [Int]$$

$$mapLength\ [\,] = [\,]$$
$$mapLength\ (x : xs) = length\ x : mapLength\ xs$$

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

Clearly there is a common denominator here of “apply some transformation to every element of a list”. And again, this design pattern can be internalized as a higher-order function, which is called *map* in the Prelude:

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

For example, we have:

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

4 Some useful functions on functions

As we’ve talked about before, functions of multiple arguments are usually defined in “curried” form in Haskell, that is, by giving them a right-nested function type – although it is equally possible to define them as functions taking a product type as input, and these two forms are equivalent. Indeed, the “currying principle” may be internalized as a higher-order function:

$$curry :: ((a, b) \to c) \to (a \to b \to c)$$
$$curry\ f\ x\ y = f\ (x, y)$$

which turns a function on pairs into a curried function, and conversely, there is an "uncurrying" principle:

$$uncurry :: (a \to b \to c) \to ((a, b) \to c)$$
$$uncurry\ g\ (x, y) = g\ x\ y$$

which turns a curried function into a function on pairs. Note that both of these may be equivalently written using lambda notation:

$$curry\ f = \backslash x \to \backslash y \to f\ (x, y)$$
$$uncurry\ g = \backslash (x, y) \to g\ x\ y$$

One reason we may want to apply currying or uncurrying in practice is to transform a function before providing it as an argument to another higher-order function. For example, $map\ (uncurry\ (+))\ [(0, 1), (2, 3), (4, 5)] = [1, 5, 9]$. Similarly to what we saw with the higher-order function *either*, the types of the currying and uncurrying transformations can be read as expressing a logical equivalence, namely that

$$(A \wedge B) \supset C \iff A \supset (B \supset C)$$

which again you can easily verify is a tautology.

Another simple but fundamental principle for defining functions is the *principle of sequential composition:* given a function $f :: a \to b$ and a function $g :: b \to c$ there is a function $h :: a \to c$ defined by $h\ x = g\ (f\ x)$. Once more, this principle may be expressed as a higher-order function:

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

(Or equivalently: $g \circ f = \backslash x \to g\ (f\ x)$.) For example, $map\ ((+1) \circ (*2))\ [0\,..\,4] = [1, 3, 5, 7, 9]$. Note that in Haskell, composition is actually notated using a period (`g . f`), although we use the notation $g \circ f$ here in these notes to emphasize the relation to the standard mathematical concept. Logically, the type of function composition corresponds to the principle of *transitivity of implication.*

Let us conclude by describing a few more important higher-order functions:[2]

(Flip the order in which a binary function takes its arguments.)

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

Example: $flip\ (-)\ 2\ 3 = 1$

(Given a value, build the constant function returning that value.)[3]

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

Example: $const\ 2\ 3 = 2$

[2]Both *flip* and *const* are already defined in the Haskell Prelude, but not *dupl*. Observe that the types of all three may be read as basic logical tautologies involving implication.

[3]Note that *const* can also be used to turn unary function into a binary function, and more generally a function of n arguments into a function of $n + 1$ arguments that ignores its first argument.

(Turn a binary function into a unary function duplicating its input.)

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

Example: $dupl\ (*)\ 3 = 9$

5 More higher-order functions on lists

The Haskell Prelude and Standard Library define a number of higher-order functions that capture common ways of manipulating lists. We describe a few of them here.

(Keep only those elements of a list satisfying a predicate.)

$$filter :: (a \to Bool) \to [a] \to [a]$$
$$filter\ p\ [\,] = [\,]$$
$$filter\ p\ (x : xs)$$
$$\quad \mid p\ x = x : filter\ p\ xs$$
$$\quad \mid otherwise = filter\ p\ xs$$

Examples:

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

(Test whether all/any elements of a list satisfy a predicate, and find an element that does.)

$$all, any :: (a \to Bool) \to [a] \to Bool$$
$$all\ p\ [\,] = True$$
$$all\ p\ (x : xs) = p\ x\ \&\&\ all\ p\ xs$$
$$any\ p\ [\,] = False$$
$$any\ p\ (x : xs) = p\ x\ ||\ any\ p\ xs$$
$$find :: (a \to Bool) \to [a] \to Maybe\ a$$
$$find\ p\ [\,] = Nothing$$
$$find\ p\ (x : xs)$$
$$\quad \mid p\ x = Just\ x$$
$$\quad \mid otherwise = find\ p\ xs$$

Examples:

```
> all (>3) [1..5]
False
> any (>3) [1..5]
True
> find (>3) [1..5]
Just 4
```

(Return the longest prefix of a list of elements that satisfy a predicate, or the remaining suffix after this prefix is dropped.)

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

Examples:

$$
\begin{aligned}
&> takeWhile\ (>3)\ [1\,..\,5] \\
&[\,] \\
&> takeWhile\ (<3)\ [1\,..\,5] \\
&[1, 2] \\
&> dropWhile\ (>3)\ [1\,..\,5] \\
&[1, 2, 3, 4, 5] \\
&> dropWhile\ (<3)\ [1\,..\,5] \\
&[3, 4, 5]
\end{aligned}
$$

(Map a function sending an element to a list over all the elements of a list, and concatenate the results.)

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

Examples:

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

Note that $concatMap\ f = concat \circ map\ f$, where $concat :: [[a]] \to [a]$ computes the concatenation of a list of lists.

The higher-order function *concatMap* may be used to understand "list comprehension" notation. For example, the expression

$$[(x, y) \mid x \leftarrow [1\,..\,3], y \leftarrow [1\,..\,3], x < y]$$

which evaluates to $[(1, 2), (1, 3), (2, 3)]$, may be equivalently expressed using *concatMap* as

$$
\begin{aligned}
&concatMap\ (\backslash x \to concatMap\ (\backslash y \to \\
&\quad \textbf{if}\ x < y\ \textbf{then}\ [(x, y)]\ \textbf{else}\ [\,])\ [1\,..\,3])\ [1\,..\,3]
\end{aligned}
$$

or equivalently (making use of the higher-order function *flip*) as

```
flip concatMap [1..3] (\x →
flip concatMap [1..3] (\y →
if x < y then [(x, y)] else []))
```

(*concatMap* is also related to the monad structure on lists, which this translation of list comprehensions implicitly relies upon. We will talk more about that in Lecture 4.)

6 *foldr*: the Swiss army knife of list functions

Remarkably, all of the higher-order functions on lists that we described in Section 5, and many other functions besides, can be defined as instances of a single higher-order function. Before jumping to the definition of this higher-order function, let's give a bit of motivation of where the definition comes from.

Suppose that we have a list and we want to extract some value from it, by writing a function

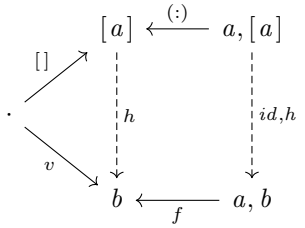
$$h :: [a] \to b$$

for some types a and b. For example, we may have a list of integers and we want to extract a boolean, i.e., $a = Int$, $b = Bool$, and $h :: [Int] \to Bool$. Since (finite) lists are built up inductively starting from the empty list, it is natural to try to define the function h itself inductively, by providing a base case for its value on the empty list, and explaining how to compute the value of the function on a non-empty list from the value of the head and the value of the function on the tail.

Here the "base case" corresponds to providing a value $v :: b$ such that $h\ [\,] = v$, while the "inductive step" corresponds to providing a function $f :: a \to b \to b$ such that $h\ (x :: xs) = f\ x\ (h\ xs)$. We thus obtain the following recursive definition of h:

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

The schema for defining h can be summarized by the following diagram, similar to the coproduct diagram we saw in Lecture 1:



Since this schema for defining h is completely generic in the "base case" v and "inductive step" f, we can internalize it as a higher-order function, which is called *foldr* in the Haskell Prelude:

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

Here are definitions for most of the functions in Section 5 in terms of *foldr*:

$$
\begin{aligned}
&\mathit{filter}\ p = \mathit{foldr}\ (\backslash x\ xs \to \textbf{if}\ p\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs)\ [\,] \\
&\mathit{all}\ p\ \ = \mathit{foldr}\ (\backslash x\ b \to p\ x\ \&\&\ b)\ \mathit{True} \\
&\mathit{any}\ p = \mathit{foldr}\ (\backslash x\ b \to p\ x\ ||\ b)\ \mathit{False} \\
&\mathit{find}\ p = \mathit{foldr}\ (\backslash x\ mx \to \textbf{if}\ p\ x\ \textbf{then}\ \mathit{Just}\ x\ \textbf{else}\ mx)\ \mathit{Nothing} \\
&\mathit{takeWhile}\ p = \mathit{foldr}\ (\backslash x\ xs \to \textbf{if}\ p\ x\ \textbf{then}\ x : xs\ \textbf{else}\ [\,])\ [\,] \\
&\mathit{concatMap}\ f = \mathit{foldr}\ (\backslash x\ ys \to f\ x +\!\!+ ys)\ [\,]
\end{aligned}
$$

And here are a few more examples of first-order functions defined by pattern-matching and recursion, and their more concise definitions in terms of *foldr*:

(Take the sum of the numbers in a list.)

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

which may be summarized as:

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

(Take the product of the numbers in a list.)

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

which may be summarized as:

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

(Return the length of a list.)

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

which may be summarized as:

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

(Return the concatenation of a list of lists.)

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

which may be summarized as:

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

(Copy a list.)

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

which may be summarized as:

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

In all of the above examples we used *foldr* to define a function taking a single list as input. However, it is possible to use *foldr* to define functions of multiple arguments, by seeing them as curried functions, and taking the target type b in the type schema

$$\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b$$

to be a function type. For example, consider the binary concatenation operation $(+\!\!+)$, which we saw how to define recursively in Lecture 1:

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

To translate this into a fold, let's first get rid of the infix notation and write $(+\!\!+)$ as a prefix operator just so as to not get distracted by syntax:

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

And then let's write the function equivalently using lambda notation, while also inserting a judicious pair of parentheses into its type:

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

We can now see how to fit $(+\!\!+)$ within the general inductive schema for defining functions $h :: [a] \to b$ described at the beginning of this section:

1. We take $b = [a] \to [a]$ as the target type.
2. We take the identity function $\backslash ys \to ys$ for the base case $v :: [a] \to [a]$.
3. We take the higher-order function $\backslash x\ g\ ys \to x : g\ ys$ for the inductive case $f :: a \to ([a] \to [a]) \to ([a] \to [a])$.

Finally, writing *id* for the identity function, and observing that

$$(\backslash x\ g\ ys \to x : g\ ys) = (\backslash x\ g \to (x:) \circ g)$$

we derive the following concise definition of $(+\!\!+)$:

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

Exercise 6.1. Use *foldr* to define a function $altsum :: Num\ a \Rightarrow [a] \to a$ which takes the alternating sum of the numbers in a list. (Example: $altsum\ [1..5] = 1 - 2 + 3 - 4 + 5 = 3$.)

Exercise 6.2. Use *foldr* to define the following functions from the standard library exported by Data.List:

1. $intersperse :: a \to [a] \to [a]$ which intersperses a given element between the elements of a list. (Example: *intersperse* `','` `"abcde"` = `"a,b,c,d,e"`.)
2. $tails :: [a] \to [[a]]$ that returns the list of all suffixes of a list. (Example: *tails* `"abcde"` = [`"abcde"`, `"bcde"`, `"cde"`, `"de"`, `"e"`, `""`].)
3. $isPrefixOf :: Eq\ a \Rightarrow [a] \to [a] \to Bool$ determining whether the first list is a prefix of the second. (Examples: *isPrefixOf* `"abc"` `"abcde"` = *True* but *isPrefixOf* `"abc"` `"def"` = *False*.)

Exercise 6.3. The astute reader will have noticed that we did not define *dropWhile* when we showed how to define the higher-order functions from Section 5 in terms of *foldr*. Nevertheless it is possible to define *dropWhile p* just using *foldr*, without using recursion. Explain how. *Hint:* Be careful not to define a variation of *filter*. You are allowed to invoke *foldr* and then do something with the result, as long as you do not use recursion. Note that there is more than one solution to this problem.

7 Aside: Folding from the left

Let's have a closer look at the way that the definition of *sum* in terms of *foldr* is used to compute the sum of the numbers 1 through 5:

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

We see that the additions are performed from right to left. In general, $foldr\ f\ v\ xs$ computes its answer by starting with the value v and combining it using f with each of the elements of xs, going right-to-left from the last element to the first.

Sometimes it can be more natural and/or more efficient to go in the other direction, applying the operation to the elements of the list from left to right. This general pattern of recursion is captured by the higher-order function *foldl*:

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

Many of the functions we defined using *foldr* in Section 6 may also be defined using *foldl*, for instance:

$$\begin{aligned}
&\mathit{sum} \quad = \mathit{foldl}\ (+)\ 0\\
&\mathit{product} = \mathit{foldl}\ (*)\ 1\\
&\mathit{concat} = \mathit{foldl}\ (+\!\!+)\ [\,]
\end{aligned}$$

These definitions are equivalent to the ones using *foldr* in the sense that they yield the same functions from inputs to output, but in fact the recursive computations are very different. For example, here is the computation of the sum of the numbers 1 through 5 using *foldl*:

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

We see that now the additions are evaluated from left to right. Of course the order of evaluation does not matter for the final result here because addition is an associative operation, although it may matter for efficiency.

Exercise 7.1. The computation of $\mathit{foldl}\ (+)\ 0\ [1,2,3,4,5]$ might remind you of the implementation of *reverse* using an accumulator that we discussed in Lecture 1. Show how to define *reverse* just using *foldl*.

Exercise 7.2. Prove that if $f :: a \to a \to a$ and $v :: a$ satisfy the equations of a monoid, then $\mathit{foldr}\ f\ v\ \mathit{xs} = \mathit{foldl}\ f\ v\ \mathit{xs}$.

8 Higher-order functions over trees

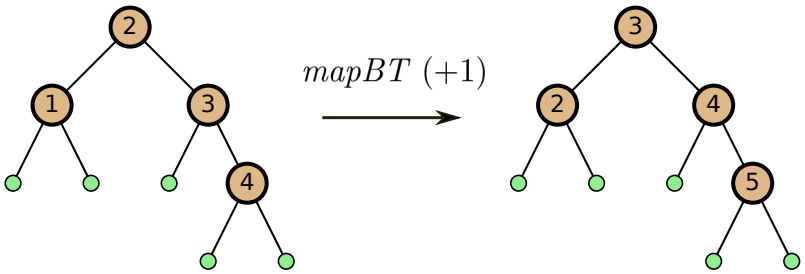
We've spent a while discussing higher-order functions over lists, but it is often useful to define higher-order functions over other data types as well. For instance, recall the data type of binary trees with labelled nodes that we introduced in Lecture 1:

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

It supports a natural analogue of the *map* function on lists, applying some transformation to every node label contained in the tree:

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

For example, we have:



There is also a natural analogue of *foldr*, which inductively defines a function $BinTree\ a \to b$ by providing a "base case" and an "inductive step":

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

For example, all of the functions we defined in Lecture 1 can be equivalently expressed as folds:

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

9 Type classes

By now we've seen several examples of polymorphic functions with type class constraints, such as

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

and so on. Intuitively, these constraints express the minimal requirements on the otherwise generic type a needed in order to define these functions – for example, *sort* requires a comparison operation on elements, *lookup* requires an equality test, and *sum* and *prod* require some notion of "number" equipped with addition/multiplication operations.

Formally, a type class is defined by specifying the type signatures of operations that need to be implemented to qualify as a member of the class, possibly together with default implementations of some of the operations in terms of the others. For example, the *Eq* class is defined as follows:

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

This says that to satisfy the *Eq* type class constraint, a type a needs to at least provide either an equality operation $(\equiv) :: a \to a \to Bool$ or an inequality

operation $(\not\equiv) :: a \to a \to \mathit{Bool}$, and if both are not provided then one is defined as the negation of the other by default. The way that we show that a type class constraint is satisfied is by providing an *instance*. For example, we can define an instance of the *Eq* class for booleans like so:

```
instance Eq Bool where
  x ≡ y = if x then y else not y
```

Sometimes we can derive a type class instance for a type constructor if we assume instances for its arguments. For example, two lists can be tested for equality provided that their underlying type of elements supports an equality test:

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

In Haskell it is also possible to express that one type class inherits operations from another type class, in a manner similar to the class hierarchy in object-oriented programming. For example, here is how the *Ord* class is defined in the Prelude (note *Ordering* is a datatype with three values *LT*, *EQ*, and *GT*):

```
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
```

The definition might look a bit complicated, but that's mainly because of all of the default implementations. To provide an instance of *Ord* it suffices to give an implementation of $(\leqslant) :: a \to a \to \mathit{Bool}$ or of $\mathit{compare} :: a \to a \to \mathit{Ordering}$, assuming you already have an instance of *Eq*.

It is often implicit in the definition of a type class that the operations obey certain laws. For example, for any instance of the *Eq* class, the binary operation $(\equiv)$ should be reflexive ($x \equiv x$), symmetric (if $x \equiv y$ then $y = x$)), and transitive (if $x \equiv y$ and $y \equiv z$ then $x \equiv z$): mathematically, we summarize this by saying that $(\equiv)$ should be an equivalence relation. Similarly, for any instance of *Ord*, the operation $(\leqslant)$ should be a linear ordering, i.e., reflexive, transitive, anti-symmetric (if $x \leqslant y$ and $y \leqslant x$ then $x \equiv y$), and total (either $x \leqslant y$ or $y \leqslant x$ for all x and y). However, although these expectations are sometimes described in the documentation of a type class, it is up to the programmer to respect them

when defining an instance of the class, as they are not enforced by the Haskell language.[4]

Finally, let's quickly have a look at the *Num* type class:

$$
\begin{aligned}
&\textbf{class } \mathit{Num}\ a \textbf{ where}\\
&\quad (+), (-), (*) :: a \to a \to a\\
&\quad \mathit{negate} \quad :: a \to a\\
&\quad \mathit{abs} \quad :: a \to a\\
&\quad \mathit{signum} \quad :: a \to a\\
&\quad \mathit{fromInteger} :: \mathit{Integer} \to a\\
&\quad x - y \quad = x + \mathit{negate}\ y\\
&\quad \mathit{negate}\ x \quad = 0 - x
\end{aligned}
$$

The operations $(+), (-), (*)$ are expected to satisfy the usual arithmetic laws (associativity, distributivity, etc.), while $\mathit{signum}\ x$ and $\mathit{abs}\ x$ are meant to return the sign and absolute value, respectively, of the value x, satisfying the equation $x = \mathit{signum}\ x * \mathit{abs}\ x$. The Haskell Prelude defines several instances of the *Num* type class, such as *Num Int*, *Num Integer*, and *Num Double*.

Type classes are a very appealing and useful feature of Haskell, but in a certain sense they may be seen as "just" a convenient mechanism for defining higher-order functions. A type class constraint in the type signature of a function may always be replaced by the types of the operations in (a minimal definition of) the corresponding type class, turning these operations into additional arguments to be passed to the function. For example, instead of the first-order function $\mathit{lookup} :: \mathit{Eq}\ a \Rightarrow a \to [(a, b)] \to \mathit{Maybe}\ b$ with a constrained polymorphic type, we can define a higher-order function

$$
\begin{aligned}
&\mathit{lookupHO} :: (a \to a \to \mathit{Bool}) \to a \to [(a, b)] \to \mathit{Maybe}\ b\\
&\mathit{lookupHO}\ \mathit{eq}\ k\ [\,] = \mathit{Nothing}\\
&\mathit{lookupHO}\ \mathit{eq}\ k\ ((k', v) : \mathit{kvs})\\
&\quad \mid \mathit{eq}\ k\ k' \quad = \mathit{Just}\ v\\
&\quad \mid \mathit{otherwise} = \mathit{lookupHO}\ \mathit{eq}\ k\ \mathit{kvs}
\end{aligned}
$$

which takes an equality operation as its first argument. Indeed, the definition of *lookupHO* is almost exactly the same as the definition of *lookup* (which we saw in Lecture 1), except that calls to the equality operation $(\equiv)$ have been replaced by calls to the first argument *eq*. Similarly, instead of defining a function $\mathit{sort} :: \mathit{Ord}\ a \Rightarrow [a] \to [a]$ with a constrained polymorphic type, we can define a higher-order sorting function

$$\mathit{sortHO} :: (a \to a \to \mathit{Bool}) \to [a] \to [a]$$

that expects a comparison operation as its first argument. Then when we want to apply *sortHO* to some list, we simply have to supply this additional comparison operation explicitly.

Still, although this translation of type class constraints into higher-order functions is semantically valid, it does have some drawbacks. Note that every call to the higher-order functions *sortHO* and *lookupHO* has to pass an extra

[4] Although they could be enforced in principle, potentially at the cost of an additional burden of proof on the part of the programmer. Indeed, dependently-typed languages such as Agda/Lean/Rocq make it possible to express, prove, and enforce such properties.

argument, corresponding to the chosen comparison/equality operation – and while in these examples there is only one additional argument, in general a type class constraint could translate to many extra arguments. One practical benefit of Haskell's type class mechanism is that these "semantically implicit" arguments are automatically inferred by the type system, by determining the types of the other arguments and examining the instances defined within a program. For example, we can run the following code in the interpreter:

```
> 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"]
```

The type checker can infer that in the first call to *sort* the input is a list of numbers while in the second it is a list of strings, and automatically chooses the appropriate instance of *Ord*. (We will talk more about type inference in Lecture 3.)

Although Haskell's approach also has drawbacks. Notably, it is only possible to define a single instance of a type class for a given type. As an example, consider the *Monoid* type class (defined in `Data.Monoid`), which expresses that a type a is equipped with a binary operation $mappend :: a \to a \to a$ and element $mempty :: a$, which moreover should be associative and unital. Concatenation and the empty list naturally give lists the structure of a monoid, and indeed the standard library defines an instance $Monoid\ [a]$ taking $mappend = (+\!\!+)$ and $mempty = [\,]$. However, there is no single canonical way of giving the integers the structure of a monoid. Indeed there are two: using addition ($mappend = (+)$, $mempty = 0$) or multiplication ($mappend = (*)$, $mempty = 1$).

One way around this limitation is with Haskell's **newtype** keyword, the purpose of which is to create an isomorphic copy of a type. Syntactically, **newtype** behaves similarly to the **data** keyword except that the right hand side of the equals sign is only allowed to contain a single constructor with a single argument. For example, the standard library introduces the following types:

```
newtype Sum a = Sum a
newtype Product a = Product a
```

The definition implies that both *Sum a* and *Product a* are isomorphic to a, but since they have different names they can be used to define different type class instances. In particular we can now define two different instances of the *Monoid* class, using addition or multiplication:

```
instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  mappend (Sum x) (Sum y) = Sum (x + y)
instance Num a => Monoid (Product a) where
  mempty = Product 1
  mappend (Product x) (Product y) = Product (x * y)
```