6.3 Combining the elements of a list: foldr

Example

\[
\begin{align*}
\text{sum} \ [\ ] &= 0 \\
\text{sum} \ (x:xs) &= x + \text{sum} \ xs
\end{align*}
\]
6.3 Combining the elements of a list: \texttt{foldr}

Example

\[
\begin{align*}
\text{sum } [] & = 0 \\
\text{sum } (x:xs) & = x + \text{sum } xs \\
\text{sum } [x_1, \ldots, x_n] & = x_1 + \ldots + x_n + 0 \\
\text{concat } [] & = [] \\
\text{concat } (xs:xs) & = xs ++ \text{concat } xs \\
\text{concat } [x_1, \ldots, x_n] & = x_1 ++ \ldots ++ x_n ++ []
\end{align*}
\]

foldr

\[
\text{foldr } (\oplus) z [x_1, \ldots, x_n] = x_1 \oplus \ldots \oplus x_n \oplus z
\]

Defined in Prelude:

\[
\begin{align*}
\text{foldr } :: (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow [a] \rightarrow a \\
\text{foldr } f a [] & = a \\
\text{foldr } f a (x:xs) & = x \ 'f' \ \text{foldr } f a \ xs
\end{align*}
\]
foldr

foldr (⊕) z [x₁, ..., xₙ] = x₁ ⊕ ... ⊕ xₙ ⊕ z

Defined in Prelude:
foldr :: (a -> a -> a) -> a -> [a] -> a
foldr f a [] = a
foldr f a (x:xs) = x 'f' foldr f a xs

Applications:
sum xs = foldr (+) 0 xs
concat xss = foldr (++) [] xss

What is the most general type of foldr?
foldr

foldr (⊕) z [x₁, ..., xₙ] = x₁ ⊕ ... ⊕ xₙ ⊕ z

Defined in Prelude:

foldr :: (a → a → a) → a → [a] → a
foldr f a [] = a
foldr f a (x:xs) = x 'f' foldr f a xs

Applications:

sum xs = foldr (+) 0 xs
concat xss = foldr (++) [] xss

What is the most general type of foldr?

Evaluating foldr

foldr f a [] = a
foldr f a (x:xs) = x 'f' foldr f a xs

Evaluating foldr

foldr f a [] = a
foldr f a (x:xs) = x 'f' foldr f a xs
foldr (+) 0 [1, -2] = foldr (+) 0 (1 : -2 : [])
Evaluating foldr

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

foldr (+) 0 [1, -2]
= foldr (+) 0 (1 : -2 : [])
= 1 + foldr (+) 0 (-2 : [])
= 1 + -2 + (foldr (+) 0 [])
= 1 + -2 + 0
= -1
More applications of foldr

\[ \text{product } xs = \text{foldr } (*) \ 1 \ xs \]

\[ \text{and } xs = \text{foldr } (\&\&) \ \text{True} \ xs \]

\[ \text{or } xs = \text{foldr } (\|\|) \ \text{False} \ xs \]

\[ \text{inSort } xs = \text{foldr } \text{ins} \ [ ] \ xs \]
What is \( \text{foldr} (:) \ ys \ xs \) ?

Example: \( \text{foldr} (:) \ ys \ (1:2:3:[]) = \)

Proof by induction on \( xs \) (Exercise!)
Defining functions via `foldr`

- means you have understood the art of higher-order functions
- allows you to apply properties of `foldr`

**Example**

If \( f \) is associative and \( a \), \( f \) \( x = x \) then

\[
foldr \ f \ a \ (xs++ys) = foldr \ f \ a \ xs \ 'f' \ foldr \ f \ a \ ys.
\]

Proof by induction on \( xs \). Induction step:

\[
foldr \ f \ a \ ((x:xs) ++ ys) = foldr \ f \ a \ (x : (xs++ys))
\]

\[
= x \ 'f' \ foldr \ f \ a \ (xs++ys)
\]

\[
= x \ 'f' \ (foldr \ f \ a \ xs \ 'f' \ foldr \ f \ a \ ys) \quad \text{by IH}
\]

\[
foldr \ f \ a \ (x:xs) \ 'f' \ foldr \ f \ a \ ys
\]

\[
= (x \ 'f' \ foldr \ f \ a \ xs) \ 'f' \ foldr \ f \ a \ ys
\]

\[
= x \ 'f' \ (foldr \ f \ a \ xs \ 'f' \ foldr \ f \ a \ ys) \quad \text{by assoc.}
\]

Therefore, if \( g \) \( xs = foldr \ f \ a \ xs \),

then \( g \ (xs ++ ys) = g \ xs \ 'f' \ g \ ys. \)
6.4 Lambda expressions

Consider

squares xs = map sq x xs where sq x = x * x

Defining functions via foldr

- means you have understood the art of higher-order functions
- allows you to apply properties of foldr

Example

If f is associative and a 'f' x = x then
foldr f a (xs++ys) = foldr f a xs 'f' foldr f a ys.

Proof by induction on xs. Induction step:
foldr f a ((x:xs) ++ ys) = foldr f a (x : (xs ++ ys))
= x 'f' foldr f a (xs ++ ys)
= x 'f' (foldr f a xs 'f' foldr f a ys) -- by IH
foldr f a (xs : xs) 'f' foldr f a ys
= (x 'f' foldr f a xs) 'f' foldr f a ys
= x 'f' (foldr f a xs 'f' foldr f a ys) -- by assoc.

Therefore, if g xs = foldr f a xs,
then g (xs ++ ys) = g xs 'f' g ys.

Therefore sum (xs ++ ys) = sum xs + sum ys.
product (xs ++ ys) = product xs * product ys,...

6.4 Lambda expressions

Consider

squares xs = map sq x xs where sq x = x * x

Do we really need to define sq explicitly? No!
6.4 Lambda expressions

Consider

squares xs = map sqr xs where sqr x = x * x

Do we really need to define sqr explicitly? No!

\( x \rightarrow x \times x \)

is the anonymous function with

formal parameter \( x \) and result \( x \times x \)

In mathematics: \( x \mapsto x \times x \)

Evaluation:

\((\lambda x \mapsto x \times x)\) 3 = 3 \times 3 = 9
6.4 Lambda expressions

Consider

\[ \text{squares } xs = \text{ map } \text{sqr } xs \text{ where } \text{sqr } x = x \times x \]

Do we really need to define \text{sqr} explicitly? No!

\[ (x \rightarrow x \times x) \]

is the anonymous function with

\[ \text{formal parameter } x \text{ and result } x \times x \]

In mathematics: \[ x \mapsto x \times x \]

Evaluation:

\[ (\langle x \rightarrow x \times x \rangle) 3 = 3 \times 3 = 9 \]

Usage:

\[ \text{squares } xs = \text{ map } (\langle x \rightarrow x \times x \rangle) xs \]

---

**Terminology**

\[ (\langle x \rightarrow e_1 \rangle) e_2 \]

\[ x: \text{ formal parameter} \]

\[ e_1: \text{ result} \]

\[ e_2: \text{ actual parameter} \]

Why “lambda”?

The logician Alonzo Church invented \textit{lambda calculus} in the 1930s

---

Logicians write \[ \lambda x. e \] instead of \[ \langle x \rightarrow e \rangle \]
Typing lambda expressions

Example
\((\lambda x . x > 0) :: \text{Int} \to \text{Bool}\)
because \(x :: \text{Int} \implies x > 0 :: \text{Bool}\)

The general rule:

\((\lambda \cdot e) :: T_1 \to T_2\)
if \(x :: T_1 \implies e :: T_2\)

Evaluating lambda expressions

\((\lambda x \cdot \text{body}) \ arg = \text{body} \text{ with } x \text{ replaced by } \arg\)

Example
\((\lambda x . x > 0) :: \text{Int} \to \text{Bool}\)
because \(x :: \text{Int} \implies x > 0 :: \text{Bool}\)
Evaluating lambda expressions

\((\lambda x \rightarrow \text{body}) \ arg = \text{body with } x \text{ replaced by } \ arg\)

Example
\((\lambda xs \rightarrow xs ++ xs) \ [1] = [1] ++ [1]\)

Sections of infix operators

\((+ 1) \text{ means } (\lambda x \rightarrow x + 1)\)

\((2 \times) \text{ means } (\lambda x \rightarrow 2 \times x)\)
Sections of infix operators

(+ 1) means \( \lambda x \rightarrow x + 1 \)
(2 *) means \( \lambda x \rightarrow 2 * x \)
(2 ~) means \( \lambda x \rightarrow 2 ^ x \)
(\ ~ 2) means \( \lambda x \rightarrow x ^ 2 \)

etc

Example
squares xs = map (\ ~ 2) xs
List comprehension

Just syntactic sugar for combinations of map

\[
[f \ x \mid x \leftarrow xs] = \text{map } f \ xs
\]

filter

\[
[x \mid x \leftarrow xs, \ p \ x] = \text{filter } \ p \ xs
\]

and concat

\[
[f \ x \ y \mid x \leftarrow xs, \ y \leftarrow ys] = \text{concat}() \ 
\]

concat (map (   ) xs)
List comprehension

Just syntactic sugar for combinations of map
\[
[ f \; x \mid x \leftarrow xs ] = \; \text{map} \; f \; xs
\]
filter
\[
[ x \mid x \leftarrow xs, \; p \; x ] = \; \text{filter} \; p \; xs
\]
and concat
\[
[f \; x \; y \mid x \leftarrow xs, \; y \leftarrow ys] = \text{concat} \; (\text{map} \; (\lambda \; x \rightarrow \text{map} \; (\lambda \; y \rightarrow \; ) \; ys) \; xs)
\]

6.5 Extensionality

Two functions are equal
if for all arguments they yield the same result

\[
f, g :: T_1 \rightarrow T:\quad \forall a. \; f \; a = g \; a
\]
\[
\frac{}{f = g}
\]
6.5 Extensionality

Two functions are equal if for all arguments they yield the same result

\[ f, g :: \ T_1 \to T : \]

\[ \forall a. \ f \ a = g \ a \]

\[ f = g \]

\[ f, g :: \ T_1 \to T_2 \to T : \]

\[ \forall a, b. \ f \ a \ b = g \ a \ b \]

\[ f = g \]

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

\[ f :: \text{Int} \to \text{Int} \to \text{Int} \]

\[ f \ x \ y = x + y \]

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

\[ f :: \text{Int} \to \text{Int} \to \text{Int} \]

\[ f \ x \ y = x + y \]

\[ f \ x = \ \lambda y \to x + y \]
6.6 Curried functions
A trick (re)invented by the logician Haskell Curry

Example

\[
\begin{align*}
  f :: \text{Int} & \to \text{Int} \to \text{Int} & f :: \text{Int} & \to (\text{Int} \to \text{Int}) \\
  f \ x \ y & = \ x + y & f \ x & = \ \lambda \ y \to \ x + y
\end{align*}
\]

Both mean the same:

\[
\begin{align*}
  f \ a \ b & = a + b \\
  (f \ a) \ b & = a + b
\end{align*}
\]
6.6 Curried functions
A trick (re)invented by the logician Haskell Curry

Example

\[ f :: \text{Int} \to \text{Int} \to \text{Int} \quad f :: \text{Int} \to (\text{Int} \to \text{Int}) \]
\[ f \ x \ y = x + y \quad f \ x = \ \lambda y \to x + y \]
Both mean the same:

\[ f \ a \ b = a + b \quad (f \ a) \ b = \ (\lambda y \to a + y) \ b = a + b \]

The trick: any function of two arguments can be viewed as a function of the first argument

In general

Every function is a function of one argument (which may return a function as a result)

\[ T_1 \to T_2 \to T \]
is just syntactic sugar for

\[ T_1 \to (T_2 \to T) \]
In general

Every function is a function of one argument (which may return a function as a result)

\[ T_1 \to T_2 \to T \]

is just syntactic sugar for

\[ T_1 \to (T_2 \to T) \]

\[ f \; e_1 \; e_2 \]
is just syntactic sugar for

\[ (f \; e_1) \; e_2 \]

Analogously for more arguments

\[ T_1 \to T_2 \to T \]

is just syntactic sugar for

\[ T_1 \to (T_2 \to T) \]

\[ f \; e_1 \; e_2 \]
is just syntactic sugar for

\[ (f \; e_1) \; e_2 \]

[\[ :: T_2 \to T \]\]

\[ \rightarrow \text{is not associative:} \]

\[ T_1 \to (T_2 \to T) \neq (T_1 \to T_2) \to T \]

Example

\[ f :: \text{Int} \to (\text{Int} \to \text{Int}) \]

\[ f \; x \; y = x + y \]
\[ T_1 \rightarrow (T_2 \rightarrow T) \neq (T_1 \rightarrow T_2) \rightarrow T \]

**Example**
\[
\begin{align*}
f &: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \\
g &: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \\
f \ x \ y &= x + y \\
g \ h &= h \ 0 + 1
\end{align*}
\]

**Quiz**

- head\-tail\-xs

Correct?

- head (tail xs)
Quiz

head tail xs
Correct?

head (tail xs)

Quiz

Partial application

Every function of \( n \) parameters can be applied to less than \( n \) arguments

Partial application

Every function of \( n \) parameters can be applied to less than \( n \) arguments

Example

Instead of \( \text{sum } xs = \text{foldr } (+) 0 \text{ } xs \) just define \( \text{sum } = \text{foldr } (+) 0 \)
Partial application

Every function of \( n \) parameters can be applied to less than \( n \) arguments

Example
Instead of \( \text{sum} \ x \text{s} = \text{foldr} \ (+) \ 0 \ x \text{s} \)
just define \( \text{sum} = \text{foldr} \ (+) \ 0 \)

In general:
If \( f :: T_1 \to \ldots \to T_n \to T \)
and \( a_1 :: T_1, \ldots, a_m :: T_m \) and \( m \leq n \)
then \( f \ a_1 \ldots a_m :: T_{m+1} \to \ldots \to T_n \to T \)

6.7 More library functions

\( \cdot \) :: (b -> c) -> (a -> b) -> (a -> c)
\( f \cdot g = \lambda x \to f (g \ x) \)
6.7 More library functions

\((.) :: (b -> c) -> (a -> b) -> (a -> c)\)
\[ f \cdot g = \\lambda x \rightarrow f \ (g \ x) \]

Example

head2 = head . tail

head2 \ [1,2,3]  
= (head . tail) \ [1,2,3]  

head2 \ [1,2,3]  
= (head . tail) \ [1,2,3]  
  = (\ \lambda x \rightarrow \text{head} \ (\text{tail} \ x)) \ [1,2,3]  
  = \text{head} \ (\text{tail} \ [1,2,3])  
  = \text{head} \ [2,3]  
  = 2
const :: a \rightarrow (b \rightarrow a)
const x = \_ \rightarrow x

const :: a \rightarrow (b \rightarrow a)
const x = \_ \rightarrow x

curry :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)
curry f = \_ x y \rightarrow f(x,y)

all :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}
all p xs = \text{and} [p x | x <- xs]

uncurry :: (a \rightarrow b \rightarrow c) \rightarrow ((a,b) \rightarrow c)
uncurry f = \_ (x,y) \rightarrow f x y
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]

Example
all (>1) [0, 1, 2] = False

any :: (a -> Bool) -> [a] -> Bool
any p = or [p x | x <- xs]

Example
any (>1) [0, 1, 2] = True

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
    | p x = x : takeWhile p xs
    | otherwise = []
6.8 Case study: Counting words

Input: A string, e.g. "never say never again"

Example

taxWhile (not . isSpace) "the end"
= "the"

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = x:xs

Example

dropWhile (not . isSpace) "the end"