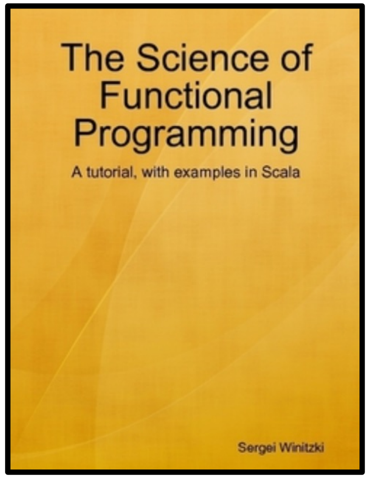
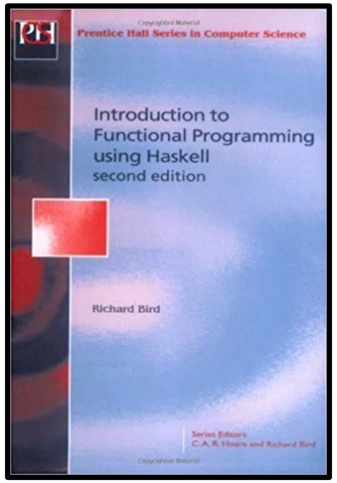
Left and Right Folds
Comparison of a mathematical definition and a programmatic one

Polyglot FP for Fun and Profit - Haskell and Scala

Based on definitions from



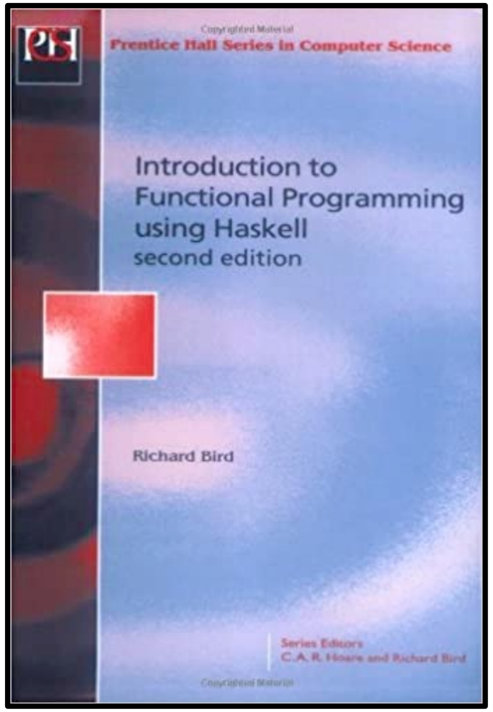


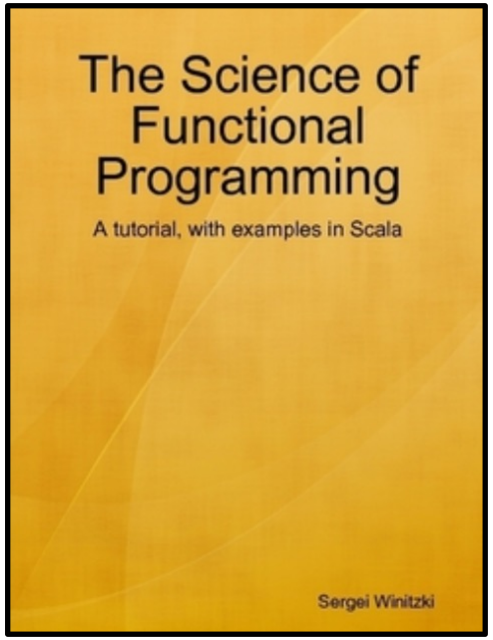
Sergei Winitzki
sergei-winitzki-11a6431

Richard Bird
http://www.cs.ox.ac.uk/people/richard.bird/

slides by

@philip_schwarz
slideshare https://www.slideshare.net/pjschwarz





@philip_schwarz

If I have to write down the definitions of a **left fold** and a **right fold** for lists, here is what I write

$$
\begin{aligned}
&foldl :: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldl\ f\ e\ [\,] = e \\
&foldl\ f\ e\ (x : xs) = foldl\ f\ (f\ e\ x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&foldr :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldr\ f\ e\ [\,] = e \\
&foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs)
\end{aligned}
$$

Richard Bird

While both definitions are **recursive**, the **left fold** is **tail recursive**, whereas the **right fold** isn't.

Although I am very familiar with the above definitions, and view them as doing a good job of explaining the two **folds**, I am always interested in alternative ways of explaining things, and so I have been looking at **Sergei Winitzki**'s mathematical definitions of **left** and **right folds**, in his upcoming book: **The Science of Functional Programming** (SOFP).

Sergei's definitions of the **folds** are in the top two rows of the following table

Definition by induction	Scala code example
$f([\,]) = b\ ;\quad f(s{+\!+}[x]) = g(f(s), x)$	`f(xs) = xs.foldLeft(b)(g)`
$f([\,]) = b\ ;\quad f([x]{+\!+}s) = g(x, f(s))$	`f(xs) = xs.foldRight(b)(g)`
$x_0 = b\ ;\quad x_{k+1} = g(x_k)$	`xs = Stream.iterate(b)(g)`
$y_0 = b\ ;\quad y_{k+1} = g(y_k, x_k)$	`ys = xs.scanLeft(b)(g)`

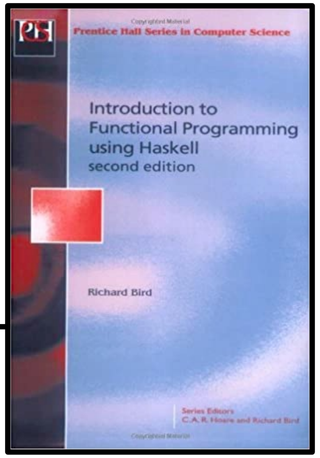
Sergei Winitzki

Left folds and right folds do not necessarily produce the same results. According to the first duality theorem of folding, one case in which the results are the same, is when we fold using the unit and associative operation of a monoid.

First duality theorem. Suppose $(\oplus)$ is associative with unit e. Then

$$foldr\ (\oplus)\ e\ xs\ =\ foldl\ (\oplus)\ e\ xs$$

For all finite lists xs.



Folding integers left and right using the (Int,+,0) monoid, for example, produces the same results.

```
test1 = TestCase (assertEqual "foldl(+) 0 []"             0 (foldl (+) 0 []))
test2 = TestCase (assertEqual "foldl(+) 0 [1,2,3,4]"     10 (foldl (+) 0 [1,2,3,4]))

test3 = TestCase (assertEqual "foldr (+) 0 []"            0 (foldr (+) 0 []))
test4 = TestCase (assertEqual "foldr (+) 0 [1,2,3,4]"    10 (foldr (+) 0 [1,2,3,4]))
```

But folding integers left and right using subtraction and zero, does not produce the same results, and in fact (Int,-,0) is not a monoid, because subtraction is not associative.

```
test5 = TestCase (assertEqual "foldl(-) 0 []"             0 (foldl (+) 0 []))
test6 = TestCase (assertEqual "foldl(-) 0 [1,2,3,4]" (- 10) (foldl (-) 0 [1,2,3,4]))

test7 = TestCase (assertEqual "foldr (-) 0 []"            0  (foldr (-) 0 []))
test8 = TestCase (assertEqual "foldr (-) 0 [1,2,3,4]"  (- 2) (foldr (-) 0 [1,2,3,4]))
```

Here are **Sergei**'s mathematical definitions again, on the right (the $+\!\!+$ operator is list concatenation). Notice how neither definition is **tail recursive**. That is deliberate.

As **Sergei** explained to me: “I'd like to avoid putting **tail recursion** into a **mathematical formula**, because **tail recursion** is just a detail of how we implement this function” and “The fact that **foldLeft** is **tail recursive** for **List** is an implementation detail that is specific to the **List** type. It will be different for other sequence types. I do not want to put the implementation details into the formulas.”

$$f([\,]) = b; \quad f(s +\!\!+ [x]) = g(f(s), x)$$

$$f([\,]) = b; \quad f([x] +\!\!+ s) = g(x, f(s))$$

To avoid any confusion (the definitions use the same function name f), and to align with the definitions on the right, let's modify **Sergei**'s definitions by doing some simple renaming.

$$\begin{aligned}
&foldl &&:: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldl\ f\ e\ [\,] &&= e \\
&foldl\ f\ e\ (x : xs) &&= foldl\ f\ (f\ e\ x)\ xs
\end{aligned}$$

$$\begin{aligned}
&foldr &&:: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldr\ f\ e\ [\,] &&= e \\
&foldr\ f\ e\ (x : xs) &&= f\ x\ (foldr\ f\ e\ xs)
\end{aligned}$$

$$f([\,]) = b; \quad f(s +\!\!+ [x]) = g(f(s), x)$$

↓

$$f \to foldl \quad g \to f \quad b \to e$$

↓

$$foldl([\,]) = e; \quad foldl(s +\!\!+ [x]) = f(foldl(s), x)$$

$$f([\,]) = b; \quad f([x] +\!\!+ s) = g(x, f(s))$$

↓

$$f \to foldr \quad g \to f \quad b \to e$$

↓

$$foldr([\,]) = e; \quad foldr([x] +\!\!+ s) = f(x, foldr(s))$$

$$foldl([\,]) = e; \quad foldl(s +\!\!+ [x]) = f(foldl(s), x)$$

$$foldr([\,]) = e; \quad foldr([x] +\!\!+ s) = f(x, foldr(s))$$

To help us understand the above two definitions, let's first express them in **Haskell**, pretending that we are able to use $s +\!\!+ [x]$ and $[x] +\!\!+ s$ in a pattern match.

```haskell
foldl f e [] = e
foldl f e (s ++ [x]) = f (foldl f e s) x
```

```haskell
foldr f e [] = e
foldr f e ([x] ++ s) = f x (foldr f e s)
```

Now let's replace $s +\!\!+ [x]$ and $[x] +\!\!+ s$ with as, and get the functions to extract the s and the x using the $head$, $tail$, $last$ and $init$. Let's also add type signatures

```haskell
foldl :: (b → a → b) → b → [a] → b
foldl f e [ ] = e
foldl f e as = f (foldl f e (init as)) (last as)
```

```haskell
foldr :: (a → b → b) → b → [a] → b
foldr f e [ ] = e
foldr f e as = f (head as)(foldr f e (tail as))
```

And now let's make it more obvious that what each **fold** is doing is taking an a from the list, **folding** the rest of the list into a b, and then returning the result of calling f with a and b.

```haskell
foldl :: (b → a → b) → b → [a] → b
foldl f e [ ] = e
foldl f e as = f b a
               where a = last as
                     b = foldl f e (init as)
```

```haskell
foldr :: (a → b → b) → b → [a] → b
foldr f e [ ] = e
foldr f e as = f a b
               where a = head as
                     b = foldr f e (tail as)
```

$$\begin{aligned}
&foldl :: (b \to a \to b) \to b \to [a] \to b\\
&foldl\ f\ e\ [\,] = e\\
&foldl\ f\ e\ as\ =\ f\ b\ a\\
&\qquad\qquad where\ a = \ last\ as\\
&\qquad\qquad\qquad b = foldl\ f\ e\ (init\ as)
\end{aligned}$$

$$\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b\\
&foldr\ f\ e\ [\,] = e\\
&foldr\ f\ e\ as\ =\ f\ a\ b\\
&\qquad\qquad where\ a = head\ as\\
&\qquad\qquad\qquad b = foldr\ f\ e\ (tail\ as)
\end{aligned}$$

Since the above two functions are very similar, let's extract their **common logic** into a **fold** function. Let's call the function that is used to extract an element from the list *take*, and the function that is used to extract the rest of the list *rest*.

$$\begin{aligned}
&fold :: (b \to a \to b) \to ([a] \to a) \to ([a] \to [a]) \to b \to [a] \to b\\
&fold\ f\ take\ rest\ e\ [\,] = e\\
&fold\ f\ take\ rest\ e\ as\ =\ f\ b\ a\\
&\qquad\qquad where\ a = take\ as\\
&\qquad\qquad\qquad b = fold\ f\ take\ rest\ e\ (rest\ as)
\end{aligned}$$

@philip_schwarz

We can now define **left** and **right folds** in terms of *fold*.

$$\begin{aligned}
&foldl :: (b \to a \to b) \to b \to [a] \to b\\
&foldl\ f\ = fold\ f\ last\ init
\end{aligned}$$

$$\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b\\
&foldr\ f\ = fold\ (flip\ f)\ head\ tail
\end{aligned}$$

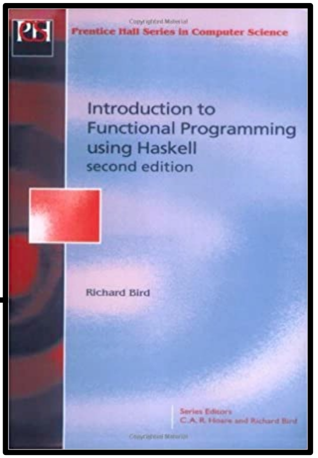
$$\begin{aligned}
&flip :: (a \to b \to c) \to b \to a \to c\\
&flip\ f\ x\ y = f\ y\ x
\end{aligned}$$

The slightly perplexing thing is that while a **left fold** applies *f* to list elements starting with the *head* of the list and proceeding from **left to right**, the above *foldl* function achieves that by navigating through list elements from **right to left**.

The fact that we can define *foldl* and *foldr* in terms of *fold*, as we do above, seems related to the **third duality theorem** of **folding**. Instead of our *foldl* function being passed the reverse of the list passed to *foldr*, it processes the list with *last* and *init*, rather than with *head* and *tail*.

Third duality theorem. For all finite lists xs,

$$\begin{aligned}
&foldr\ f\ e\ xs\ = foldl\ (flip\ f)\ e\ (reverse\ xs)\\
&\qquad\qquad \textbf{where}\ flip\ f\ x\ y = f\ y\ x
\end{aligned}$$



To summarise what we did to help understand **SOFP**'s mathematical definitions of **left and right fold**: we turned them into code and expressed them in terms of a common function $fold$ that uses a $take$ function to extract an element from the list being folded, and a $rest$ function to extract the rest of the list.

$$foldl([\,]) = e; \quad foldl(s + [x]) = f(foldl(s), x)$$

$$foldr([\,]) = e; \quad foldr([x] + s) = f(x, foldr(s))$$

$foldl :: (b \to a \to b) \to b \to [a] \to b$
$foldl\ f\ = fold\ f\ last\ init$

$fold :: (b \to a \to b) \to ([a] \to a) \to ([a] \to [a]) \to b \to [a] \to b$
$fold\ f\ take\ rest\ e\ [\,] = e$
$fold\ f\ take\ rest\ e\ as\ =\ f\ b\ a$
$\quad where\ a = take\ as$
$\quad\quad b = fold\ f\ take\ rest\ e\ (rest\ as)$

$foldr :: (a \to b \to b) \to b \to [a] \to b$
$foldr\ f\ = fold\ (flip\ f)\ head\ tail$

Let's feed one aspect of the above back into **Sergei**'s definitions. Let's temporarily rewrite them by replacing $s + [x]$ and $[x] + s$ with as, and getting the definitions to extract the s and the x using the functions $head$, $tail$, $last$ and $init$.

Notice how the flipping of f done by the $foldr$ function above, is reflected, in the $foldr$ function below, in the fact that its f takes an a and a b, rather than a b and an a.

$$foldl([\,]) = e; \quad foldl(as) = f(foldl(init(as)), last(as))$$

$$foldr([\,]) = e; \quad foldr(as) = f(head(as), foldr(tail(as)))$$

Another thing we can do to understand **SOFP**'s definitions of **left** and **right folds**, is to see how they work when applied to a sample list, e.g. $[x_0, x_1, x_2, x_3]$, when we run them manually.

In the next slide we first do this for the following definitions

$$
\begin{aligned}
&foldl && :: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldl\ f\ e\ [\,] && = e \\
&foldl\ f\ e\ (x : xs) && = foldl\ f\ (f\ e\ x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&foldr && :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldr\ f\ e\ [\,] && = e \\
&foldr\ f\ e\ (x : xs) && = f\ x\ (foldr\ f\ e\ xs)
\end{aligned}
$$

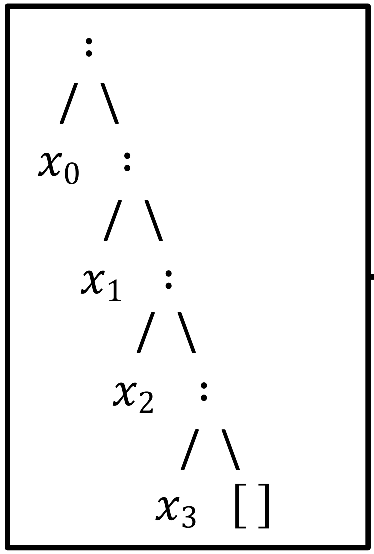
In the slide after that, we do it for **SOFP**'s definitions.

$$foldl([\,]) = e; \quad foldl(s +\!\!+ [x]) = f(foldl(s), x)$$

$$foldr([\,]) = e; \quad foldr([x] +\!\!+ s) = f(x, foldr(s))$$

$$
\begin{aligned}
&foldr &&:: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldr\ f\ e\ [\,] &&= e \\
&foldr\ f\ e\ (x : xs) &&= f\ x\ (foldr\ f\ e\ xs)
\end{aligned}
$$

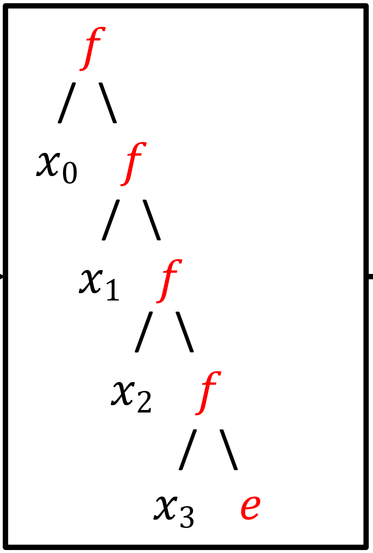
$xs = [x_0, x_1, x_2, x_3]$



$x_0 : (x_1 : (x_2 : (x_3 : [\,])))$

replace:
$:$ with f
$[\,]$ with e

$foldr\ f\ e\ xs$

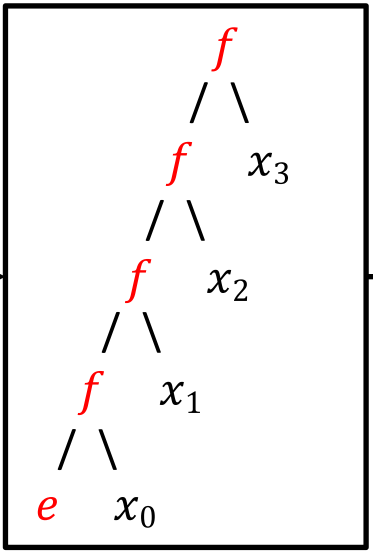


$f\ x_0\ (f\ x_1\ (f\ x_2\ (f\ x_3\ e)))$

$$
\begin{aligned}
&foldr\ f\ e\ [x_0, x_1, x_2, x_3] \\
&f\ x_0\ (foldr\ f\ e\ [x_1, x_2, x_3]) \\
&f\ x_0\ (f\ x_1\ (foldr\ f\ e\ [x_2, x_3])) \\
&f\ x_0\ (f\ x_1\ (f\ x_2\ (foldr\ f\ e\ [x_3]))) \\
&f\ x_0\ (f\ x_1\ (f\ x_2\ (f\ x_3\ (foldr\ f\ e\ [\,])))) \\
&f\ x_0\ (f\ x_1\ (f\ x_2\ (f\ x_3\ e)))
\end{aligned}
$$

$foldl\ f\ e\ xs$

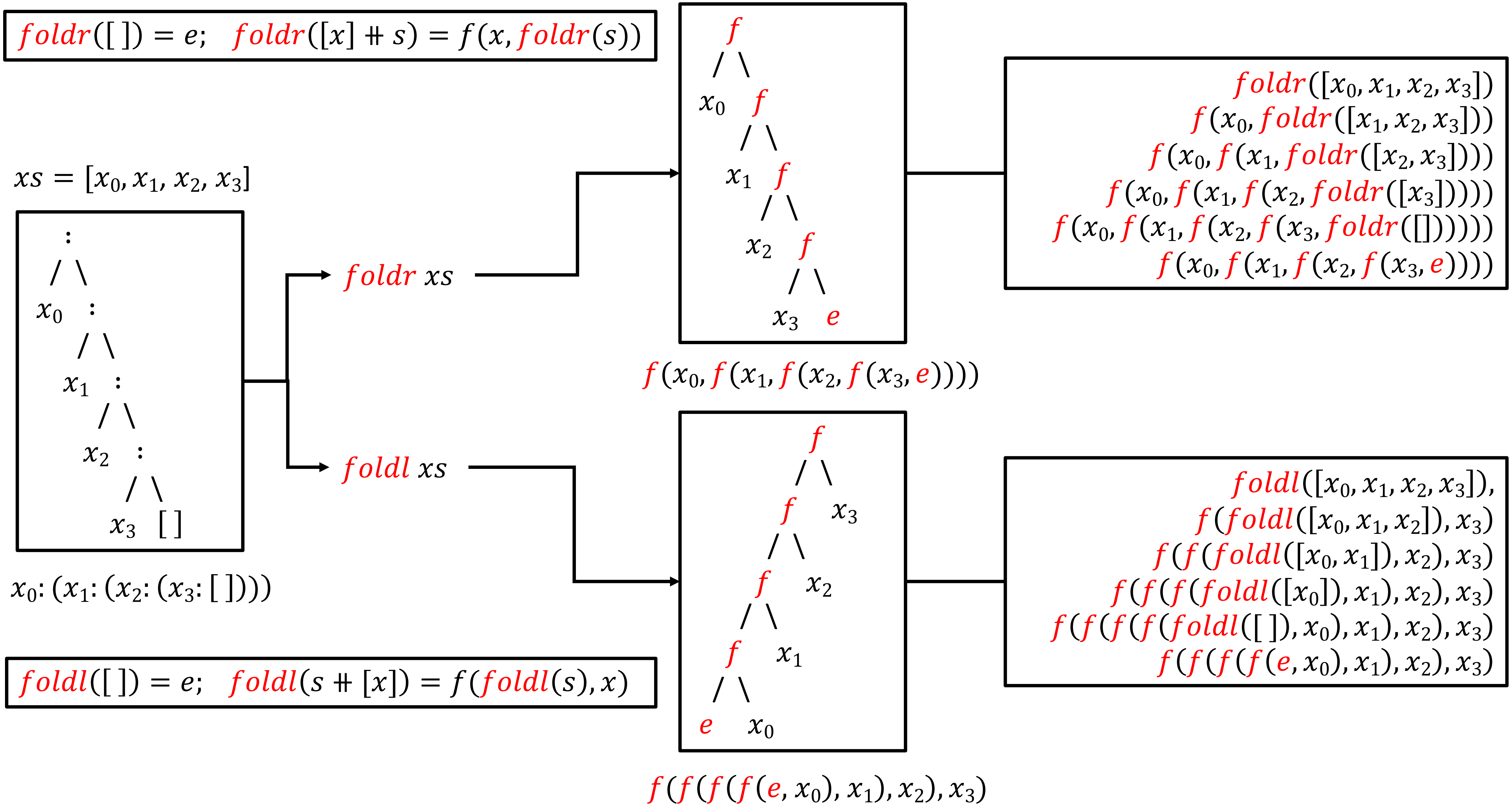
```
var acc = e
foreach(x in xs)
  acc = f(acc, x)
return acc
```



$f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)\ x_3$

$$
\begin{aligned}
&foldl\ f\ e\ [x_0, x_1, x_2, x_3] \\
&foldl\ f\ (f\ e\ x_0)[x_1, x_2, x_3] \\
&foldl\ f\ (f\ (f\ e\ x_0)\ x_1)[x_2, x_3] \\
&foldl\ f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)[x_3] \\
&foldl\ f\ (f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)\ x_3)[\,] \\
&f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)\ x_3
\end{aligned}
$$

$$
\begin{aligned}
&foldl &&:: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldl\ f\ e\ [\,] &&= e \\
&foldl\ f\ e\ (x : xs) &&= foldl\ f\ (f\ e\ x)\ xs
\end{aligned}
$$



$foldr([\,]) = e; \quad foldr([x] + s) = f(x, foldr(s))$
$xs = [x_0, x_1, x_2, x_3]$
$x_0 : (x_1 : (x_2 : (x_3 : [\,])))$
$foldr\ xs$
$foldl\ xs$
$f(x_0, f(x_1, f(x_2, f(x_3, e))))$
$foldr([x_0, x_1, x_2, x_3])$
$f(x_0, foldr([x_1, x_2, x_3]))$
$f(x_0, f(x_1, foldr([x_2, x_3])))$
$f(x_0, f(x_1, f(x_2, foldr([x_3]))))$
$f(x_0, f(x_1, f(x_2, f(x_3, foldr([\,])))))$
$f(x_0, f(x_1, f(x_2, f(x_3, e))))$
$f(f(f(f(e, x_0), x_1), x_2), x_3)$
$foldl([x_0, x_1, x_2, x_3]),$
$f(foldl([x_0, x_1, x_2]), x_3)$
$f(f(foldl([x_0, x_1]), x_2), x_3)$
$f(f(f(foldl([x_0]), x_1), x_2), x_3)$
$f(f(f(f(foldl([\,]), x_0), x_1), x_2), x_3)$
$f(f(f(f(e, x_0), x_1), x_2), x_3)$
$foldl([\,]) = e; \quad foldl(s + [x]) = f(foldl(s), x)$

@philip_schwarz

Now let's compare the results for both definitions of *foldl*. The results are the same.

$xs = [x_0, x_1, x_2, x_3]$

```
  :
 / \
x0   :
    / \
   x1   :
       / \
      x2   :
          / \
         x3  [ ]
```

$x_0 : (x_1 : (x_2 : (x_3 : [\,])))$

$$
\begin{aligned}
&foldl && :: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldl\ f\ e\ [\,] && = e \\
&foldl\ f\ e\ (x : xs) && = foldl\ f\ (f\ e\ x)\ xs
\end{aligned}
$$

$$foldl([\,]) = e; \quad foldl(s +\!\!+ [x]) = f(foldl(s), x)$$

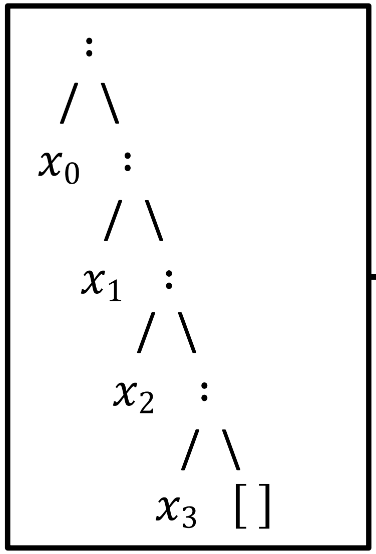
$$
\begin{aligned}
&foldl\ f\ e\ [x_0, x_1, x_2, x_3] \\
&foldl\ f\ (f\ e\ x_0)[x_1, x_2, x_3] \\
&foldl\ f\ (f\ (f\ e\ x_0)\ x_1)[x_2, x_3] \\
&foldl\ f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)[x_3] \\
&foldl\ f\ (f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)\ x_3)[\,] \\
&f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)\ x_3
\end{aligned}
$$

```
            f
           / \
          f   x3
         / \
        f   x2
       / \
      f   x1
     / \
    e   x0
```

$$
\begin{aligned}
&foldl([x_0, x_1, x_2, x_3]), \\
&f(foldl([x_0, x_1, x_2]), x_3) \\
&f(f(foldl([x_0, x_1]), x_2), x_3) \\
&f(f(f(foldl([x_0]), x_1), x_2), x_3) \\
&f(f(f(f(foldl([\,]), x_0), x_1), x_2), x_3) \\
&f(f(f(f(e, x_0), x_1), x_2), x_3)
\end{aligned}
$$

And now let's compare the results for both definitions of *foldr*. Again, the results are the same.

$xs = [x_0, x_1, x_2, x_3]$

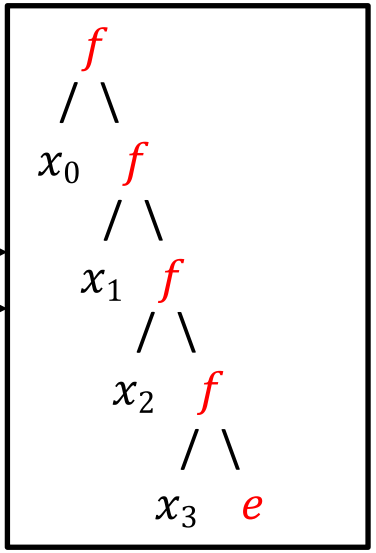


$x_0 : (x_1 : (x_2 : (x_3 : [\,])))$

$$
\begin{aligned}
&foldr :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
&foldr\ f\ e\ [\,] = e \\
&foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs)
\end{aligned}
$$

$$foldr([\,]) = e; \quad foldr([x] +\!\!+\ s) = f(x, foldr(s))$$

$$
\begin{aligned}
&foldr\ f\ e\ [x_0, x_1, x_2, x_3] \\
&f\ x_0\ (foldr\ f\ e\ [x_1, x_2, x_3]) \\
&f\ x_0\ (f\ x_1\ (foldr\ f\ e\ [x_2, x_3])) \\
&f\ x_0\ (f\ x_1\ (f\ x_2\ (foldr\ f\ e\ [x_3]))) \\
&f\ x_0\ (f\ x_1\ (f\ x_2\ (f\ x_3\ (foldr\ f\ e\ [\,])))) \\
&f\ x_0\ (f\ x_1\ (f\ x_2\ (f\ x_3\ e)))
\end{aligned}
$$



$$
\begin{aligned}
&foldr([x_0, x_1, x_2, x_3]) \\
&f(x_0, foldr([x_1, x_2, x_3])) \\
&f(x_0, f(x_1, foldr([x_2, x_3]))) \\
&f(x_0, f(x_1, f(x_2, foldr([x_3])))) \\
&f(x_0, f(x_1, f(x_2, f(x_3, foldr([\,]))))) \\
&f(x_0, f(x_1, f(x_2, f(x_3, e))))
\end{aligned}
$$

The way $foldr$ applies f to $[x, y, z]$ is by associating to the right.

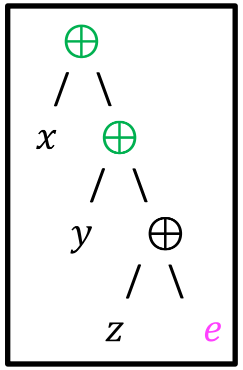
The way $foldl$ does it is by **associating** to the right.

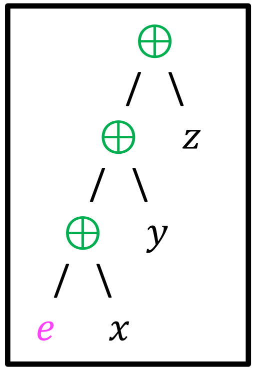
Thinking of f as an infix operator can help to grasp this.

$foldr$	$f(x, f(y, z))$	$x \oplus (y \oplus z)$
$foldl$	$f(f(x, y), z)$	$(x \oplus y) \oplus z$

$$foldr\ (\oplus)\ e\ [x, y, z] = x \oplus (y \oplus (z \oplus e))$$

$$foldl\ (\oplus)\ e\ [x, y, z] = ((e \oplus x) \oplus y) \oplus z$$





Addition is **associative**, so associating to the left and to the right yields the same result. But subtraction isn't **associative**, so associating to the left yields a result that is different to the one yielded when associating to the right.

$$foldr\ (+)\ 0\ [1,2,3] = 1 + (2 + (3 + 0)) = 6$$
$$foldl\ (+)\ 0\ [1,2,3] = ((0 + 1) + 2) + 3 = 6$$

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

We have seen that Sergei's definitions of left and right folds make perfect sense.

Not only are they simple and succinct, they are also free of implementation details like tail recursion.

That's all. I hope you found this slide deck useful.

By the way, in case you are interested, see below for a whole series of slide decks dedicated to folding.