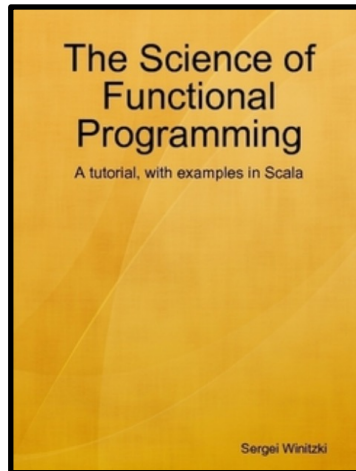


Left and Right Folds

Comparison of a mathematical definition and a programmatic one

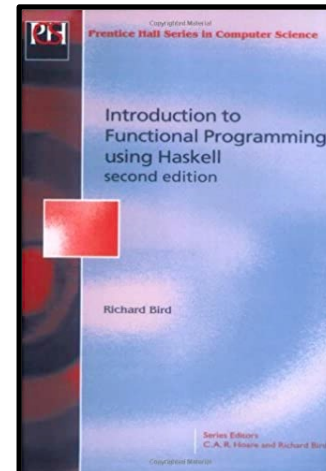
Polyglot FP for Fun and Profit - Haskell and Scala

Based on definitions from



Sergei Winitzki

 [sergei-winitzki-11a6431](https://www.linkedin.com/in/sergei-winitzki-11a6431)



Richard Bird

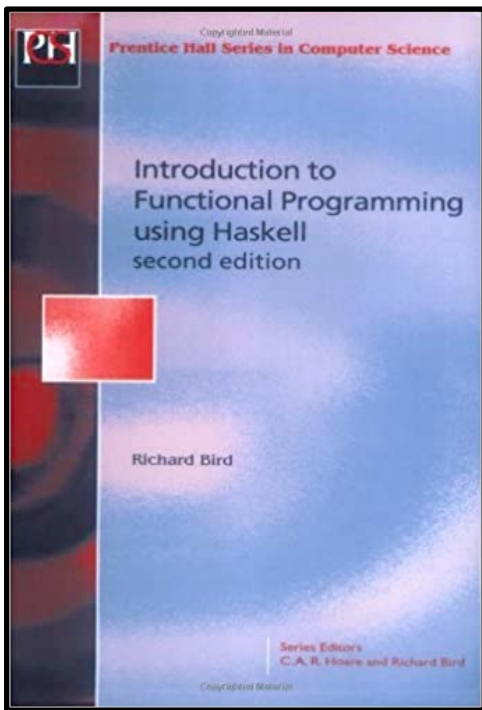
<http://www.cs.ox.ac.uk/people/richard.bird/>

slides by



 [@philip_schwarz](https://twitter.com/philip_schwarz)

 [slideshare https://www.slideshare.net/pjschwarz](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

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

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

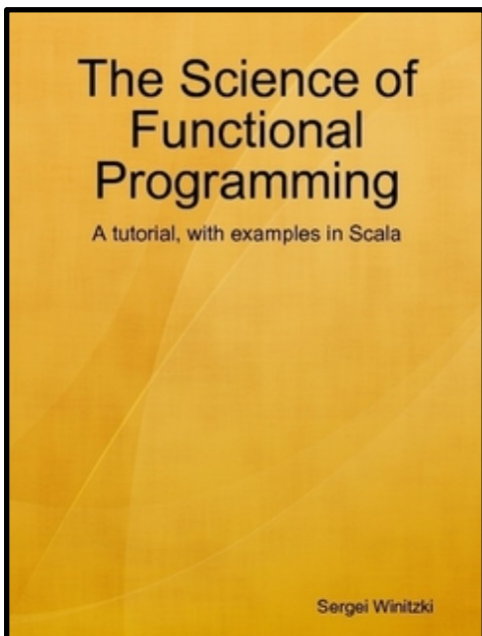

Richard Bird

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

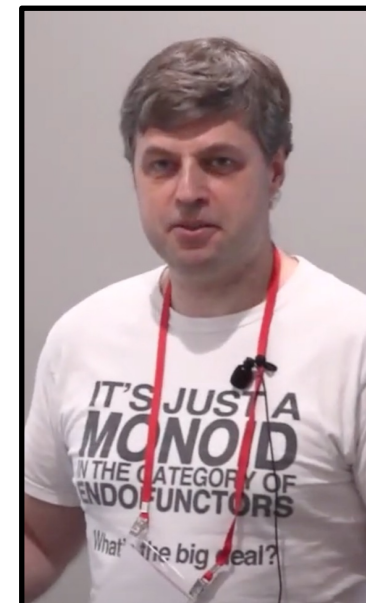
Although I am very familiar with the above definitions, and view them as doing a good job of explaining the two **folds**, I am always interested in alternative ways of explaining things, and so I have been looking at **Sergei Winitzki's** mathematical definitions of **left** and **right folds**, in 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
$f([]) = b; f(s++[x]) = g(f(s), x)$	<code>f(xs) = xs.foldLeft(b)(g)</code>
$f([]) = b; f([x]++s) = g(x, f(s))$	<code>f(xs) = xs.foldRight(b)(g)</code>
$x_0 = b; x_{k+1} = g(x_k)$	<code>xs = Stream.iterate(b)(g)</code>
$y_0 = b; y_{k+1} = g(y_k, x_k)$	<code>ys = xs.scanLeft(b)(g)</code>



Sergei Winitzki

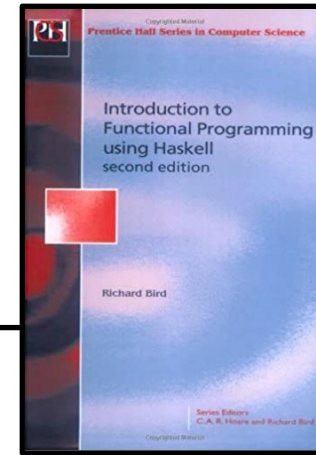


Left folds and **right folds** do not necessarily produce the same results. According to the **first duality theorem of folding**, one case in which the results are the same, is when we **fold** using the **unit** and **associative operation** of a **monoid**.

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

$$\text{foldr } (\oplus) e xs = \text{foldl } (\oplus) e xs$$

For all **finite** lists xs .



Folding integers **left** and **right** using the $(\text{Int}, +, 0)$ **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 $(\text{Int}, -, 0)$ 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."

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

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

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

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



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.

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

$$f \rightarrow \mathit{foldl} \quad g \rightarrow f \quad b \rightarrow e$$

$$\mathit{foldl}([]) = e; \quad \mathit{foldl}(s \# [x]) = f(\mathit{foldl}(s), x)$$

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

$$f \rightarrow \mathit{foldr} \quad g \rightarrow f \quad b \rightarrow e$$

$$\mathit{foldr}([]) = e; \quad \mathit{foldr}([x] \# s) = f(x, \mathit{foldr}(s))$$

$$\mathit{foldl}([\])=e; \mathit{foldl}(s \# [x])=f(\mathit{foldl}(s),x)$$

$$\mathit{foldr}([\])=e; \mathit{foldr}([x] \# s)=f(x,\mathit{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.

$$\begin{aligned}\mathit{foldl} f e [\] &= e \\ \mathit{foldl} f e (s \# [x]) &= f(\mathit{foldl} f e s) x\end{aligned}$$

$$\begin{aligned}\mathit{foldr} f e [\] &= e \\ \mathit{foldr} f e ([x] \# s) &= f x (\mathit{foldr} f e s)\end{aligned}$$



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

$$\begin{aligned}\mathit{foldl} &:: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ \mathit{foldl} f e [\] &= e \\ \mathit{foldl} f e as &= f(\mathit{foldl} f e (\mathit{init} as)) (\mathit{last} as)\end{aligned}$$

$$\begin{aligned}\mathit{foldr} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ \mathit{foldr} f e [\] &= e \\ \mathit{foldr} f e as &= f(\mathit{head} as)(\mathit{foldr} f e (\mathit{tail} as))\end{aligned}$$



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 .

$$\begin{aligned}\mathit{foldl} &:: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ \mathit{foldl} f e [\] &= e \\ \mathit{foldl} f e as &= f b a \\ &\quad \text{where } a = \mathit{last} as \\ &\quad \quad b = \mathit{foldl} f e (\mathit{init} as)\end{aligned}$$

$$\begin{aligned}\mathit{foldr} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ \mathit{foldr} f e [\] &= e \\ \mathit{foldr} f e as &= f a b \\ &\quad \text{where } a = \mathit{head} as \\ &\quad \quad b = \mathit{foldr} f e (\mathit{tail} as)\end{aligned}$$

$$\begin{aligned} \text{foldl} &:: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ \text{foldl } f \ e \ [] &= e \\ \text{foldl } f \ e \ as &= f \ b \ a \\ &\text{where } a = \text{last } as \\ &\quad b = \text{foldl } f \ e \ (\text{init } as) \end{aligned}$$

$$\begin{aligned} \text{foldr} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ \text{foldr } f \ e \ [] &= e \\ \text{foldr } f \ e \ as &= f \ a \ b \\ &\text{where } a = \text{head } as \\ &\quad b = \text{foldr } f \ e \ (\text{tail } as) \end{aligned}$$

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*.

$$\begin{aligned} \text{fold} &:: (b \rightarrow a \rightarrow b) \rightarrow ([a] \rightarrow a) \rightarrow ([a] \rightarrow [a]) \rightarrow b \rightarrow [a] \rightarrow b \\ \text{fold } f \ \text{take } \text{rest} \ e \ [] &= e \\ \text{fold } f \ \text{take } \text{rest} \ e \ as &= f \ b \ a \\ &\text{where } a = \text{take } as \\ &\quad b = \text{fold } f \ \text{take } \text{rest} \ e \ (\text{rest } as) \end{aligned}$$

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

 @philip_schwarz

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

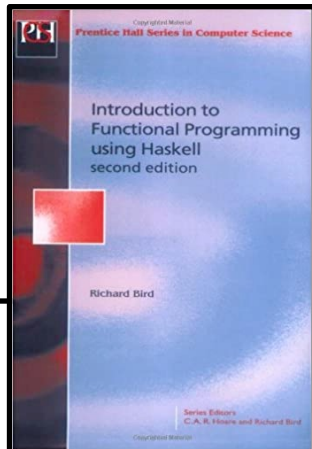
$$\begin{aligned} \text{foldr} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ \text{foldr } f &= \text{fold } (\text{flip } f) \ \text{head } \text{tail} \end{aligned}$$

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

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*,

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




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.

$$\mathit{foldl}([\] = e; \mathit{foldl}(s \# [x]) = f(\mathit{foldl}(s), x)$$

$$\mathit{foldr}([\] = e; \mathit{foldr}([x] \# s) = f(x, \mathit{foldr}(s))$$

$$\mathit{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$$

$$\mathit{foldl} f = \mathit{fold} f \mathit{last} \mathit{init}$$

$$\mathit{fold} :: (b \rightarrow a \rightarrow b) \rightarrow ([a] \rightarrow a) \rightarrow ([a] \rightarrow [a]) \rightarrow b \rightarrow [a] \rightarrow b$$

$$\mathit{fold} f \mathit{take} \mathit{rest} e [\] = e$$

$$\mathit{fold} f \mathit{take} \mathit{rest} e \mathit{as} = f b a$$

where $a = \mathit{take} \mathit{as}$
 $b = \mathit{fold} f \mathit{take} \mathit{rest} e (\mathit{rest} \mathit{as})$

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

$$\mathit{foldr} f = \mathit{fold} (\mathit{flip} f) \mathit{head} \mathit{tail}$$


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*.

$$\mathit{foldl}([\] = e; \mathit{foldl}(\mathit{as}) = f(\mathit{foldl}(\mathit{init}(\mathit{as})), \mathit{last}(\mathit{as}))$$

$$\mathit{foldr}([\] = e; \mathit{foldr}(\mathit{as}) = f(\mathit{head}(\mathit{as}), \mathit{foldr}(\mathit{tail}(\mathit{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

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

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



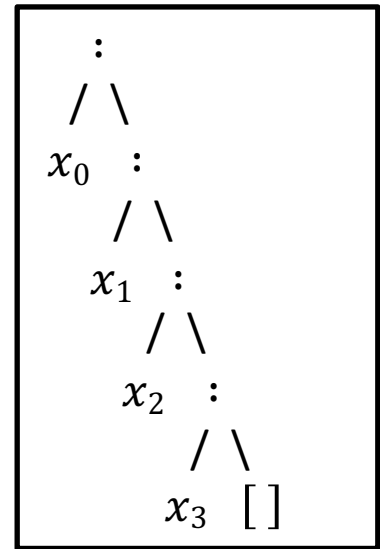
In the slide after that, we do it for **SOFP**'s definitions.

$$\mathit{foldl}([], e) = e; \quad \mathit{foldl}(s \# [x], f) = f(\mathit{foldl}(s), x)$$

$$\mathit{foldr}([], e) = e; \quad \mathit{foldr}([x] \# s, f) = f(x, \mathit{foldr}(s))$$

foldr :: ($\alpha \rightarrow \beta \rightarrow \beta$) $\rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$
foldr *f* *e* [] = *e*
foldr *f* *e* (x:xs) = *f* x (*foldr* *f* *e* xs)

xs = [x₀, x₁, x₂, x₃]



x₀: (x₁: (x₂: (x₃: [])))

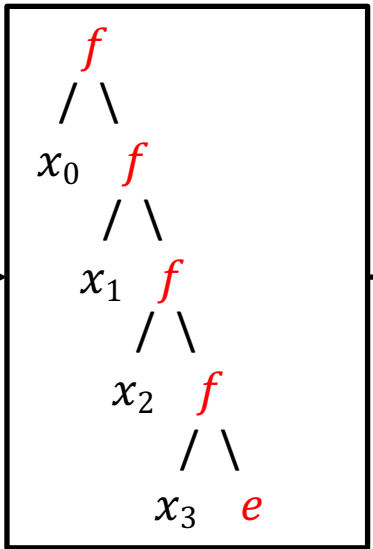
replace:
 : with *f*
 [] with *e*

foldr *f* *e* xs

foldl *f* *e* xs

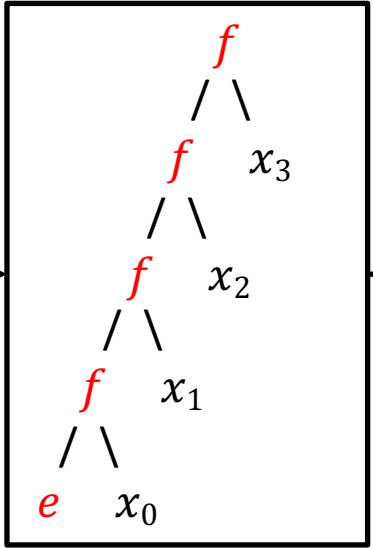
var acc = *e*
 foreach(x in xs)
 acc = *f*(acc, x)
 return acc

foldl :: ($\beta \rightarrow \alpha \rightarrow \beta$) $\rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$
foldl *f* *e* [] = *e*
foldl *f* *e* (x:xs) = *foldl* *f* (*f* *e* x) xs



f x₀ (*f* x₁ (*f* x₂ (*f* x₃ *e*)))

foldr *f* *e* [x₀, x₁, x₂, x₃]
f x₀ (*foldr* *f* *e* [x₁, x₂, x₃])
f x₀ (*f* x₁ (*foldr* *f* *e* [x₂, x₃]))
f x₀ (*f* x₁ (*f* x₂ (*foldr* *f* *e* [x₃])))
f x₀ (*f* x₁ (*f* x₂ (*f* x₃ (*foldr* *f* *e* []))))
f x₀ (*f* x₁ (*f* x₂ (*f* x₃ *e*)))

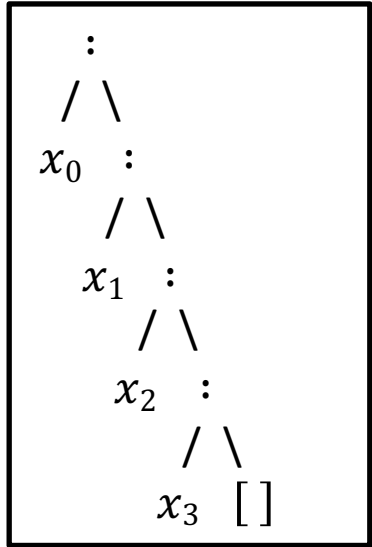


f (*f* (*f* (*f* *e* x₀) x₁) x₂) x₃

foldl *f* *e* [x₀, x₁, x₂, x₃]
foldl *f* (*f* *e* x₀) [x₁, x₂, x₃]
foldl *f* (*f* (*f* *e* x₀) x₁) [x₂, x₃]
foldl *f* (*f* (*f* (*f* *e* x₀) x₁) x₂) [x₃]
foldl *f* (*f* (*f* (*f* (*f* *e* x₀) x₁) x₂) x₃) []
f (*f* (*f* (*f* *e* x₀) x₁) x₂) x₃

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

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

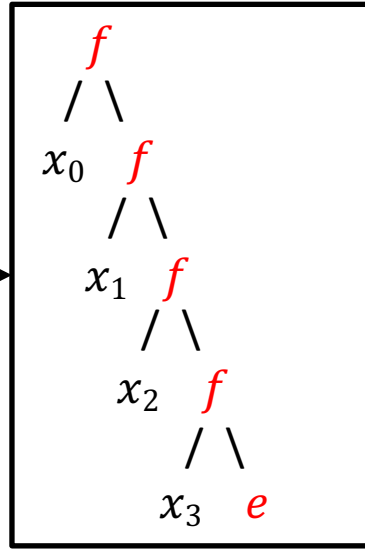


$x_0:(x_1:(x_2:(x_3:[])))$

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

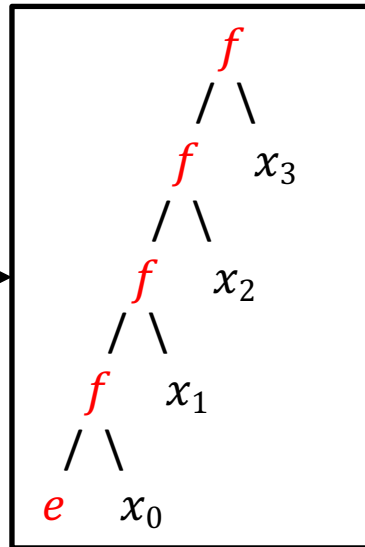
$foldr\ xs$

$foldl\ xs$



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

$foldr([x_0, x_1, x_2, x_3])$
 $f(x_0, foldr([x_1, x_2, x_3]))$
 $f(x_0, f(x_1, foldr([x_2, x_3])))$
 $f(x_0, f(x_1, f(x_2, foldr([x_3])))$
 $f(x_0, f(x_1, f(x_2, f(x_3, foldr([]))))$
 $f(x_0, f(x_1, f(x_2, f(x_3, e))))$



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

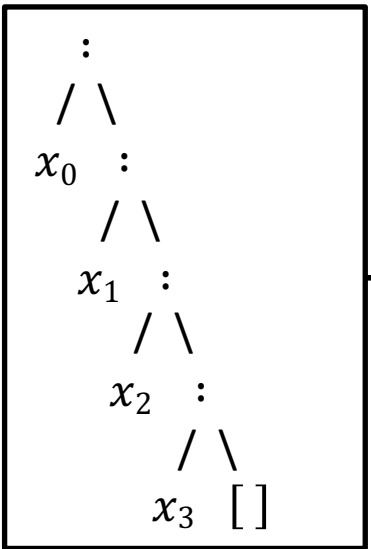
$foldl([x_0, x_1, x_2, x_3],$
 $f(foldl([x_0, x_1, x_2]), x_3)$
 $f(f(foldl([x_0, x_1]), x_2), x_3)$
 $f(f(f(foldl([x_0]), x_1), x_2), x_3)$
 $f(f(f(f(foldl([], x_0), x_1), x_2), x_3)$
 $f(f(f(f(e, x_0), x_1), x_2), x_3)$



@philip_schwarz

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

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

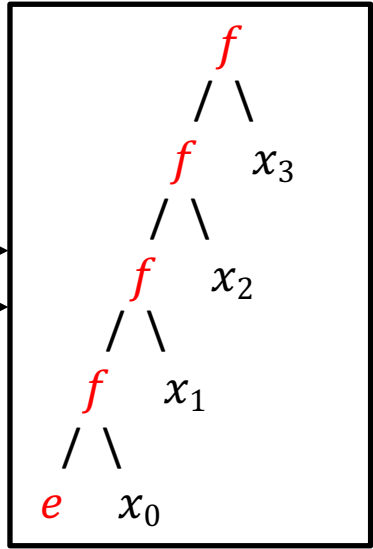


$x_0 : (x_1 : (x_2 : (x_3 : [])))$

$foldl :: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$
 $foldl\ f\ e\ [] = e$
 $foldl\ f\ e\ (x:xs) = foldl\ f\ (f\ e\ x)\ xs$

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

$foldl\ f\ e\ [x_0, x_1, x_2, x_3]$
 $foldl\ f\ (f\ e\ x_0)\ [x_1, x_2, x_3]$
 $foldl\ f\ (f\ (f\ e\ x_0)\ x_1)\ [x_2, x_3]$
 $foldl\ f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)\ [x_3]$
 $foldl\ f\ (f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)\ x_3)\ []$
 $f\ (f\ (f\ (f\ e\ x_0)\ x_1)\ x_2)\ x_3$

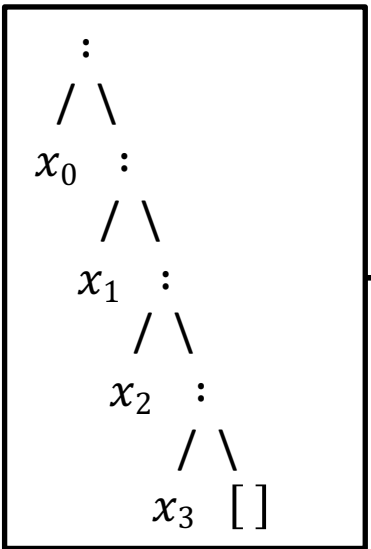


$foldl([x_0, x_1, x_2, x_3],$
 $f(foldl([x_0, x_1, x_2]), x_3)$
 $f(f(foldl([x_0, x_1]), x_2), x_3)$
 $f(f(f(foldl([x_0]), x_1), x_2), x_3)$
 $f(f(f(f(foldl([],) x_0), x_1), x_2), x_3)$
 $f(f(f(f(e, x_0), x_1), x_2), x_3)$



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

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

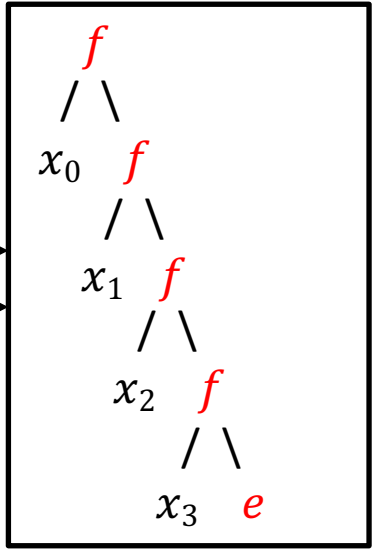


$x_0:(x_1:(x_2:(x_3:[])))$

foldr $:: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$
foldr $f e [] = e$
foldr $f e (x:xs) = f x (foldr f e xs)$

foldr $([]) = e$; *foldr* $([x] \# s) = f(x, foldr(s))$

foldr $f e [x_0, x_1, x_2, x_3]$
 $f x_0 (foldr f e [x_1, x_2, x_3])$
 $f x_0 (f x_1 (foldr f e [x_2, x_3]))$
 $f x_0 (f x_1 (f x_2 (foldr f e [x_3])))$
 $f x_0 (f x_1 (f x_2 (f x_3 (foldr f e []))))$
 $f x_0 (f x_1 (f x_2 (f x_3 e)))$



foldr $([x_0, x_1, x_2, x_3])$
 $f(x_0, foldr([x_1, x_2, x_3]))$
 $f(x_0, f(x_1, foldr([x_2, x_3])))$
 $f(x_0, f(x_1, f(x_2, foldr([x_3])))$
 $f(x_0, f(x_1, f(x_2, f(x_3, foldr([])))))$
 $f(x_0, f(x_1, f(x_2, f(x_3, e))))$



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

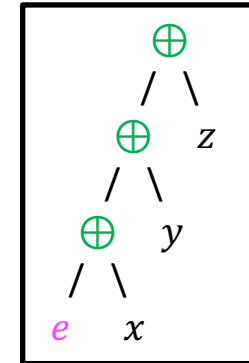
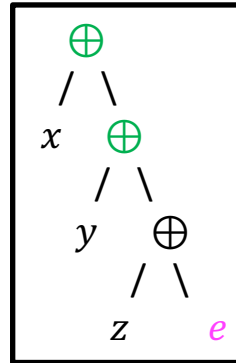
The way *foldl* does it is by **associating** to the right.

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

$$\begin{array}{ll} \textit{foldr} & f(x, f(y, z)) \quad x \oplus (y \oplus z) \\ \textit{foldl} & f(f(x, y), z) \quad (x \oplus y) \oplus z \end{array}$$

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

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



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.

$$\begin{array}{l} \textit{foldr} (+) 0 [1,2,3] = 1 + (2 + (3 + 0)) = 6 \\ \textit{foldl} (+) 0 [1,2,3] = ((0 + 1) + 2) + 3 = 6 \end{array}$$

$$\begin{array}{l} \textit{foldr} (-) 0 [1,2,3] = 1 - (2 - (3 - 0)) = 2 \\ \textit{foldl} (-) 0 [1,2,3] = ((0 - 1) - 2) - 3 = -6 \end{array}$$



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.

Folding Unfolded

Polyglot FP for Fun and Profit
Haskell and Scala

See how **recursive functions** and **structural induction** relate to **recursive datatypes**
Follow along as the **fold abstraction** is introduced and explained
Watch as **folding** is used to simplify the definition of **recursive functions** over **recursive datatypes**

Part 1 - through the work of

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON
University of Glasgow, Glasgow, UK
<http://www.cs.gla.ac.uk/~ah2/>

Richard Bird <http://www.cs.ox.ac.uk/people/richard.bird/>
Graham Hutton [@haskellhutt](http://haskellhutt.com/)




slides by [@philip_schwarz](https://www.slideshare.net/pschwarz) [slideshare https://www.slideshare.net/pschwarz](https://www.slideshare.net/pschwarz)

Folding Unfolded

Polyglot FP for Fun and Profit
Haskell and Scala

See **aggregation functions** defined **inductively** and implemented using **recursion**
Learn how in many cases, **tail-recursion** and the **accumulator trick** can be used to avoid **stackoverflow errors**
Watch as **general aggregation** is implemented and see **duality theorems** capturing the relationship between **left folds** and **right folds**

Part 2 - through the work of

Sergei Winitzki [sergei-winitzki-11a6431](http://www.cs.ox.ac.uk/people/sergei.winitzki/)
Richard Bird <http://www.cs.ox.ac.uk/people/richard.bird/>


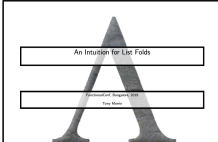


slides by [@philip_schwarz](https://www.slideshare.net/pschwarz) [slideshare https://www.slideshare.net/pschwarz](https://www.slideshare.net/pschwarz)

Folding Unfolded

Polyglot FP for Fun and Profit
Haskell and Scala

Develop the **correct intuitions** of what **fold left** and **fold right** actually do, and how different these two functions are
Learn other important concepts about **folding**, thus reinforcing and expanding on the material seen in parts 1 and 2
Includes a brief introduction to (or refresher of) **asymptotic analysis** and **θ -notation**

Part 3 - through the work of

Tony Morris [@dibblego](https://presentations.tmorris.net/) <https://presentations.tmorris.net/>
Richard Bird <http://www.cs.ox.ac.uk/people/richard.bird/>


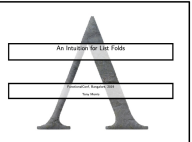


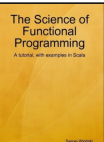

slides by [@philip_schwarz](https://www.slideshare.net/pschwarz) [slideshare https://www.slideshare.net/pschwarz](https://www.slideshare.net/pschwarz)

Folding Unfolded

Polyglot FP for Fun and Profit
Haskell and Scala

Can a **left fold** ever work over an **infinite list**? What about a **right fold**? Find out.
Learn about the other two functions used by functional programmers to implement **mathematical induction**: **iterating** and **scanning**.
Learn about the limitations of the **accumulator technique** and about **tupling**, a technique that is the dual of the **accumulator trick**.

Part 4 - through the work of

Tony Morris [@dibblego](https://presentations.tmorris.net/) <https://presentations.tmorris.net/>
Richard Bird <http://www.cs.ox.ac.uk/people/richard.bird/>
Sergei Winitzki [sergei-winitzki-11a6431](http://www.cs.ox.ac.uk/people/sergei.winitzki/)

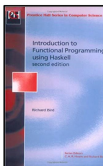








slides by [@philip_schwarz](https://www.slideshare.net/pschwarz) [slideshare https://www.slideshare.net/pschwarz](https://www.slideshare.net/pschwarz)

Folding Unfolded

Polyglot FP for Fun and Profit
Haskell and Scala

gain a deeper understanding of why **right folds** over very large and **infinite lists** are sometimes possible in Haskell
see how **lazy evaluation** and **function strictness** affect **left** and **right folds** in Haskell
learn when an ordinary **left fold** results in a **space leak** and how to avoid it using a **strict left fold**

Part 5 - through the work of

Richard Bird <http://www.cs.ox.ac.uk/people/richard.bird/>
Tony Morris [@dibblego](https://presentations.tmorris.net/) <https://presentations.tmorris.net/>
Bryan O'Sullivan, John Goerzen, Donald Bruce Stewart
Sergei Winitzki [sergei-winitzki-11a6431](http://www.cs.ox.ac.uk/people/sergei.winitzki/)
Graham Hutton [@haskellhutt](http://haskellhutt.com/)

slides by [@philip_schwarz](https://www.slideshare.net/pschwarz) [slideshare https://www.slideshare.net/pschwarz](https://www.slideshare.net/pschwarz)