Folding

$$
\begin{array}{ccccccccc}
 & : & & & & & & & \\
 & / & \backslash & & & & & & \\
a_0 & & : & & & & & & \\
 & & / & \backslash & & & & & \\
 & a_1 & & : & & & & & \\
 & & & / & \backslash & & & & \\
 & & a_2 & & : & & & & \\
 & & & & / & \backslash & & & \\
 & & & a_3 & & [\,] & & &
\end{array}
\longrightarrow
\begin{array}{ccccccccc}
 & f & & & & & & & \\
 & / & \backslash & & & & & & \\
a_0 & & f & & & & & & \\
 & & / & \backslash & & & & & \\
 & a_1 & & f & & & & & \\
 & & & / & \backslash & & & & \\
 & & a_2 & & f & & & & \\
 & & & & / & \backslash & & & \\
 & & & a_3 & & e & & &
\end{array}
$$

CHEAT-SHEET

#6

slides by  @philip_schwarz FP Illuminated https://fpilluminated.com/

$foldr :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$
$foldr\ f\ b\ [\,] = b$
$foldr\ f\ b\ (x : xs) = f\ x\ (foldr\ f\ b\ xs)$

$foldr$ is *not* **tail recursive**. ☩

Folding a sufficiently large list with $foldr$ results in a **stack overflow** error. †

$foldl :: (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta$
$foldl\ f\ b\ [\,] = b$
$foldl\ f\ b\ (x : xs) = foldl\ f\ (f\ b\ x)\ xs$

$foldl$ *is* **tail recursive**.

Folding a list (however large) with $foldl$ does not result in a **stack overflow** error. ‡

† See slide after next for an exception in **Scala**
‡ See next slide for an exception in **Haskell**
☩ See slides five to nine for a refresher on tail recursion

$$foldr :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$$
$$foldr\ f\ b\ [\,] = b$$
$$foldr\ f\ b\ (x{:}xs) = f\ x\ (foldr\ f\ b\ xs)$$

$$foldl :: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$$
$$foldl\ f\ b\ [\,] = b$$
$$foldl\ f\ b\ (x{:}xs) = foldl\ f\ (f\ b\ x)\ xs$$

Haskell

```
> :{
| foldRight :: (α -> β -> β) -> β -> [α] -> β
| foldRight f e []     = e
| foldRight f e (x:xs) = f x (foldRight f e xs)
| :}

> foldRight (+) 0 [1..10000000]
50000005000000

> foldRight (+) 0 [1..100000000]
*** Exception: stack overflow

> -- same again but using built-in function

> foldr (+) 0 [1..10000000]
50000005000000

> foldr (+) 0 [1..100000000]
*** Exception: stack overflow
```

```
> :{
| foldLeft :: (β -> α -> β) -> β -> [α] -> β
| foldLeft f e []     = e
| foldLeft f e (x:xs) = foldLeft f (f e x) xs
| :}

> foldLeft (+) 0 [1..10000000]
50000005000000

> foldLeft (+) 0 [1..100000000]
*** Exception: stack overflow

> -- same again but using built-in function

> foldl (+) 0 [1..10000000]
50000005000000

> foldl (+) 0 [1..100000000]
*** Exception: stack overflow

> Data.List.foldl' (+) 0 [1..100000000]
5000000050000000
```

These **stack overflows** have to do with **Haskell**'s **nonstrict evaluation**, and are avoided using a **strict left fold**, called **foldl'** (see final slides).

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

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

```
scala> def foldr[A,B](f: A=>B=>B)(e:B)(s:List[A]):B =
|    s match { case Nil    => e
|              case x::xs => f(x)(foldr(f)(e)(xs)) }

scala> def `(+)`: Long => Long => Long =
     |   m => n => m + n

scala> foldr(`(+)`)(0)(List(1,2,3,4))
val res1: Long = 10

scala> foldr(`(+)`)(0)(List.range(1,10_001))
val res2: Long = 50005000

scala> foldr(`(+)`)(0)(List.range(1,100_001))
java.lang.StackOverflowError

scala> // same again but using built-in function

scala> List.range(1,10_001).foldRight(0)(_+_)
val res3: Int = 50005000

scala> List.range(1,100_001).foldRight(0L)(_+_)
val res4: Long = 5000050000
```

```
scala> import scala.annotation.tailrec

scala> @tailrec
|   def foldl[A,B](f: B=>A=>B)(e:B)(s:List[A]):B =
|     s match { case Nil    => e
|               case x::xs => foldl(f)(f(e)(x))(xs) }

scala> def `(+)`: Long => Long => Long =
     |   m => n => m + n

scala> foldl(`(+)`)(0)(List.range(1,10_001))
val res1: Long = 50005000

scala> foldl(`(+)`)(0)(List.range(1,100_001))
val res2: Long = 5000050000

scala> // same again but using built-in function

scala> List.range(1,10_001).foldLeft(0)(_+_)
val res3: Int = 50005000

scala> List.range(1,100_001).foldLeft(0L)(_+_)
val res4: Long = 5000050000
```

The reason a **stack overflow** is not happening here is because built-in **foldRight** is defined in terms of **foldLeft**! (see cheat-sheet #7)

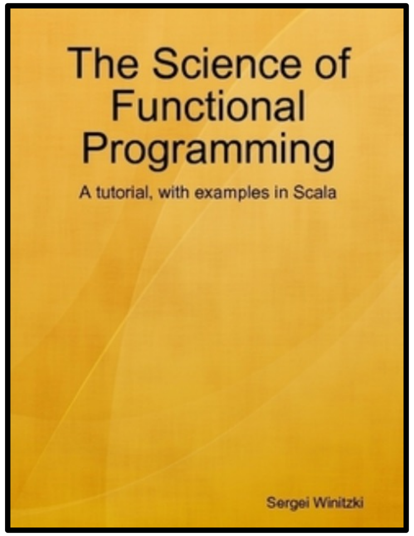
2.2.3 Tail recursion

The code of `lengthS` will fail for large enough sequences. To see why, consider an **inductive definition** of the `.length` method as a function `lengthS`:

```
def lengthS(s: Seq[Int]): Int =
  if (s.isEmpty) 0
  else 1 + lengthS(s.tail)

scala> lengthS((1 to 1000).toList)
  res0: Int = 1000

scala> val s = (1 to 100_000).toList
s : List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, ...

scala> lengthS(s)
java.lang.StackOverflowError
at .lengthS(<console>:12)
at .lengthS(<console>:12)
at .lengthS(<console>:12)
at .lengthS(<console>:12)
...
```

The problem is not due to insufficient main memory: we *are* able to compute and hold in memory the entire sequence `s`. The problem is with the code of the function `lengthS`. This function calls itself inside the expression `1 + lengthS(...)`. Let us visualize how the computer evaluates that code:

```
lengthS(Seq(1, 2, ..., 100_000))
= 1 + lengthS(Seq(2, ..., 100_000))
= 1 + (1 + lengthS(Seq(3, ..., 100_000)))
= ...
```



Sergei Winitzki

in sergei-winitzki-11a6431

```scala
def lengthS(s: Seq[Int]): Int =
  if (s.isEmpty) 0
  else 1 + lengthS(s.tail)
```

```scala
lengthS(Seq(1, 2, ..., 100_000))
= 1 + lengthS(Seq(2, ..., 100_000))
= 1 + (1 + lengthS(Seq(3, ..., 100_000)))
= ...
```

The code of lengthS will evaluate the inductive step, that is, the “else” part of the “if/else”, about 100,000 times. Each time, the intermediate sub-expression with nested computations `1+(1+(...))` will get larger.

That sub-expression needs to be held somewhere in memory, until the function body goes into the base case with no more recursive calls. When that happens, the intermediate sub-expression will contain about 100_000 nested function calls still waiting to be evaluated.

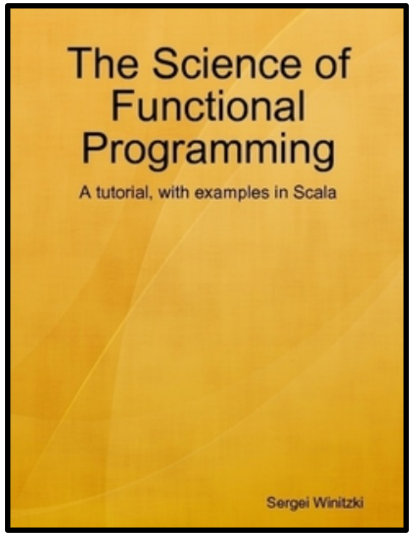
A special area of memory called stack memory is dedicated to storing the arguments for all not-yet-evaluated nested function calls. Due to the way computer memory is managed, the stack memory has a fixed size and cannot grow automatically. So, when the intermediate expression becomes large enough, it causes an overflow of the stack memory and crashes the program.

One way to avoid stack overflows is to use a trick called tail recursion. Using tail recursion means rewriting the code so that all recursive calls occur at the end positions (at the “tails”) of the function body. In other words, each recursive call must be *itself* the last computation in the function body, rather than placed inside other computations. Here is an example of tail-recursive code:

```scala
def lengthT(s: Seq[Int], res: Int): Int =
  if (s.isEmpty) res
  else lengthT(s.tail, res + 1)
```

In this code, one of the branches of the if/else returns a fixed value without doing any recursive calls, while the other branch returns the result of a recursive call to `lengthT(...)`.

It is not a problem that the recursive call to `lengthT` has some sub-expressions such as `res + 1` as its arguments, because all these sub-expressions will be computed *before* `lengthT` is recursively called.



Sergei Winitzki

in sergei-winitzki-11a6431

The recursive call to lengthT is the last computation performed by this branch of the if/else. A **tail-recursive** function can have many if/else or match/case branches, with or without **recursive calls**; but **all recursive calls must be always the last expressions returned**.

The **Scala** compiler will always use tail recursion when possible. Additionally, **Scala** has a feature for verifying that a function's code is **tail-recursive**: the `tailrec` **annotation**. If a function with a `tailrec` **annotation** is not **tail-recursive** (or is not **recursive** at all), the program will not compile. The code of `lengthT` with a `tailrec` annotation looks like this:

```scala
import scala.annotation.tailrec

@tailrec def lengthT(s: Seq[Int], res: Int): Int =
if (s.isEmpty) res
else lengthT(s.tail, res + 1)
```

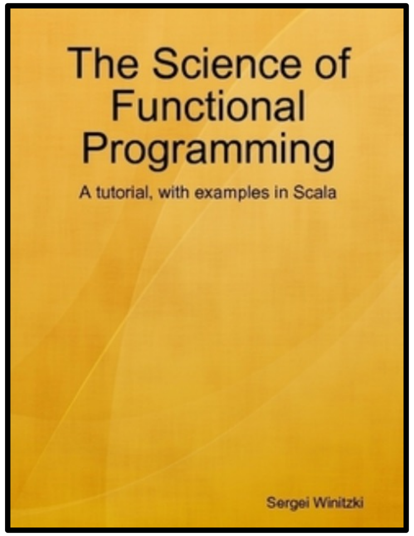
```scala
def lengthS(s: Seq[Int]): Int =
  if (s.isEmpty) 0
  else 1 + lengthS(s.tail)
```

Let us trace the evaluation of this function on an example:

```scala
  lengthT(Seq(1,2,3), 0)
= lengthT(Seq(2,3), 0 + 1) // = lengthT(Seq(2,3), 1)
= lengthT(Seq(3), 1 + 1)   // = lengthT(Seq(3), 2)
= lengthT(Seq(), 2 + 1)    // = lengthT(Seq(), 3)
= 3
```

All sub-expressions such as `1 + 1` **and** `2 + 1` **are computed _before_ recursive calls to** `lengthT`. **Because of that, sub-expressions do not grow within the stack memory**. **This is the main benefit of tail recursion**.

How did we rewrite the code of `lengthS` into the **tail-recursive** code of `lengthT`? An important difference between `lengthS` and `lengthT` is the additional argument (**res**), called the **accumulator argument**. **This argument is equal to an intermediate result of the computation**. The next **intermediate result** (`res + 1`) is computed and passed on to the next **recursive call** via the **accumulator argument**. In the **base case** of the **recursion**, the function now returns the **accumulated result** (**res**) rather than 0, because at that time the computation is finished. **Rewriting code by adding an accumulator argument to achieve tail recursion is called the accumulator technique or the "accumulator trick"**.



Sergei Winitzki

sergei-winitzki-11a6431

One consequence of using the **accumulator trick** is that the function `lengthT` now always needs a value for the **accumulator argument**. However, our goal is to implement a function such as `length(s)` with just one argument, `s:Seq[Int]`. We can define `length(s) = lengthT(s, ???)` if we supply an initial **accumulator value**. The correct initial value for the **accumulator** is 0, since in the **base case** (an **empty sequence s**) we need to return 0.

It appears useful to define the helper function (`lengthT`) separately. Then `length` will just call `lengthT` and specify the initial value of the **accumulator argument**. To emphasize that `lengthT` is a helper function that is only used by `length` to achieve **tail recursion**, we define `lengthT` as a nested function inside the code of `length`:

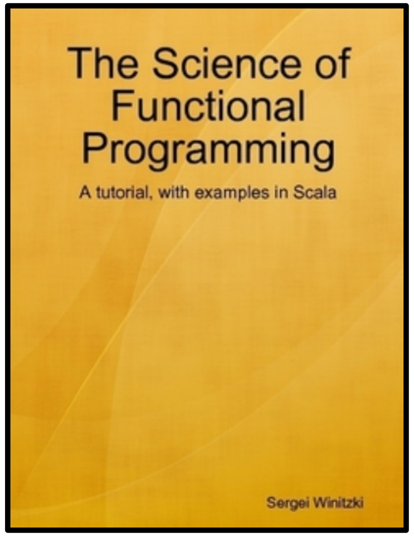
```
import scala.annotation.tailrec

def length[A](xs: Seq[A]): Int = {
  @tailrec def lengthT(s: Seq[A], res: Int): Int = {
    if (s.isEmpty) res
    else lengthT(s.tail, res + 1)
  }
  lengthT(xs, 0)
}
```

When `length` is implemented like that, users will not be able to call `lengthT` directly, because `lengthT` is only visible within the body of the `length` function. Another possibility in **Scala** is to use a **default value** for the **res** argument:

```
@tailrec def length(s: Seq[A], res: Int = 0): Int =
  if (s.isEmpty) res
  else length(s.tail, res + 1)
```

Giving a default value for a function argument is the same as defining *two* functions: one with that argument and one without. For example, the syntax

```
def f(x: Int, y: Boolean = false): Int = ... // Function body.
```



Sergei Winitzki
sergei-winitzki-11a6431

is equivalent to defining two functions with the same name but different numbers of arguments:

```
def f(x: Int, y: Boolean) = ...    // Define the function body here.
def f(x: Int): Int = f(Int, false) // Call the function defined above.
```

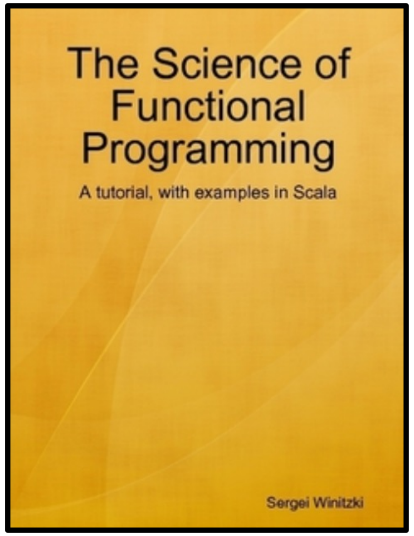
Using a default argument value, we can define the tail-recursive helper function and the main function at once, making the code shorter.

The accumulator trick works in a large number of cases, but it may be not obvious how to introduce the accumulator argument, what its initial value must be, and how to define the inductive step for the accumulator. In the example with the `lengthT` function, the accumulator trick works because of the following mathematical property of the expression being computed:

$$1 + (1 + (1 + (\ldots + 0))) \;=\; (((0 + 1) + 1) + \ldots) + 1\,.$$

This equation follows from the associativity law of addition. So, the computation can be rearranged to group all additions to the left. During the evaluation, the accumulator's value corresponds to a certain number of left-grouped parentheses $((0 + 1) \ldots) + 1$. In code, it means that intermediate expressions are fully computed before making recursive calls; So, recursive calls always occur outside all other sub-expressions - that is, in tail positions. There are no sub-expressions that need to be stored on the stack until all the recursive calls are complete.

However, not all computations can be rearranged in that way. Even if a code rearrangement exists, it may not be immediately obvious how to find it.



Sergei Winitzki

in sergei-winitzki-11a6431

Left Folds, Laziness, and Space Leaks

To keep our initial discussion simple, we use `foldl` throughout most of this section. This is convenient for testing, but we will never use `foldl` in practice. The reason has to do with **Haskell**'s **nonstrict evaluation**. **If we apply `foldl` (+) [1,2,3], it evaluates to the expression** `(((0 + 1) + 2) + 3)`. We can see this occur if we revisit the way in which the function gets expanded:

```
    foldl (+) 0                   (1:2:3:[])
 == foldl (+) (0 + 1)             (2:3:[])
 == foldl (+) ((0 + 1) + 2)       (3:[])
 == foldl (+) (((0 + 1) + 2) + 3) []
 ==           (((0 + 1) + 2) + 3)
```

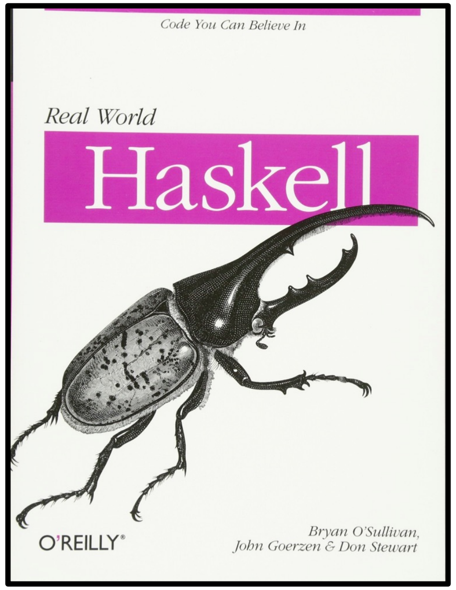
The final expression will not be evaluated to 6 until its value is demanded. Before it is evaluated, it must be stored as a thunk. Not surprisingly, a thunk is more expensive to store than a single number, and the more complex the thunked expression, the more space it needs. For something cheap such as arithmetic, thunking an expression is more computationally expensive than evaluating it immediately. We thus end up paying both in space and in time.
When GHC is evaluating a thunked expression, it uses an internal stack to do so. Because a thunked expression could potentially be infinitely large, GHC places a fixed limit on the maximum size of this stack. Thanks to this limit, we can try a large thunked expression in *ghci* without needing to worry that it might consume all the memory:

```
ghci> foldl (+) 0 [1..1000]
500500
```

From looking at this expansion, we can surmise that **this creates a thunk that consists of** 1,000 **integers and** 999 **applications of** (+). **That's a lot of memory and effort to represent a single number! With a larger expression, although the size is still modest, the results are more dramatic**:

```
ghci> foldl (+) 0 [1..1000000]
*** Exception: stack overflow
```

On small expressions, `foldl` will work correctly but slowly, due to the thunking overhead that it incurs.



Bryan O'Sullivan
John Goerzen
Donald Bruce Stewart

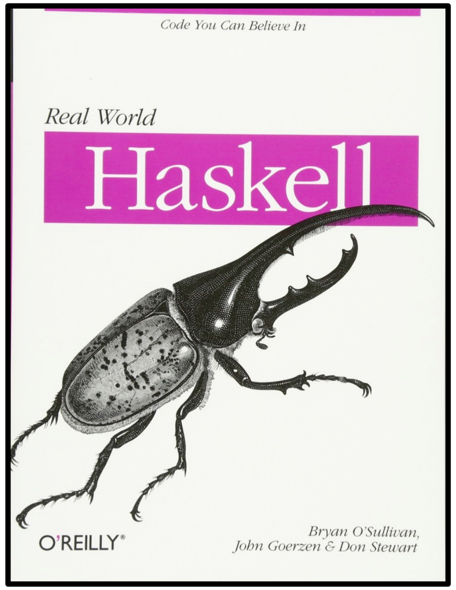
We refer to this invisible thunking as a *space leak*, because our code is operating normally, but it is using far more memory than it should.

On larger expressions, code with a space leak will simply fail, as above. A space leak with foldl is a classic roadblock for new Haskell programmers. Fortunately, this is easy to avoid.

The `Data.List` module defines a function named foldl' that is similar to foldl, but does not build up thunks. The difference in behavior between the two is immediately obvious:

```
ghci> foldl (+) 0 [1..1000000]
*** Exception: stack overflow
ghci> :module +Data.List
ghci> foldl' (+) 0 [1..1000000]
500000500000
```

Due to `foldl`'s thunking behavior, it is wise to avoid this function in real programs, even if it doesn't fail outright, it will be unnecessarily inefficient. Instead, import `Data.List` and use `foldl'`.



Bryan O'Sullivan
John Goerzen
Donald Bruce Stewart

based on

Folding Unfolded

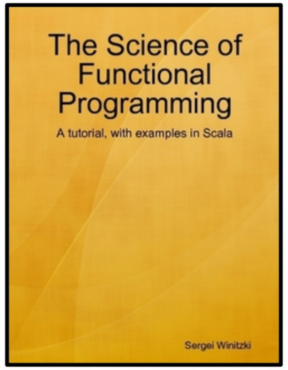
Polyglot FP for Fun and Profit
Haskell and Scala

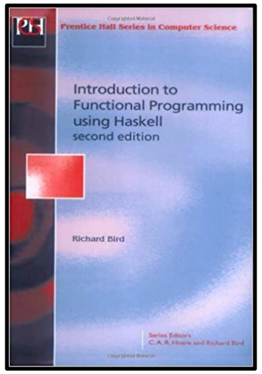
See **aggregation functions** defined **inductively** and implemented using **recursion**

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

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

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

FP Illuminated https://fpilluminated.com/