

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.

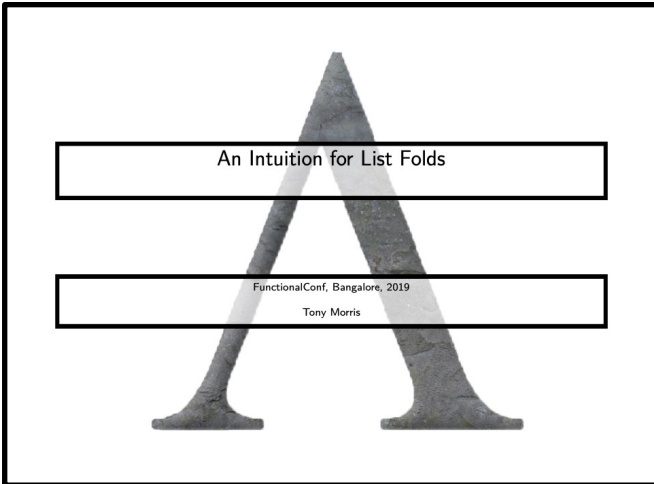
Learn about the other two functions used by functional programmers to implement **mathematical induction**: **iterating** and **scanning**.

Learn about the limitations of the **accumulator technique** and about **tupling**, a technique that is the dual of the **accumulator trick**.

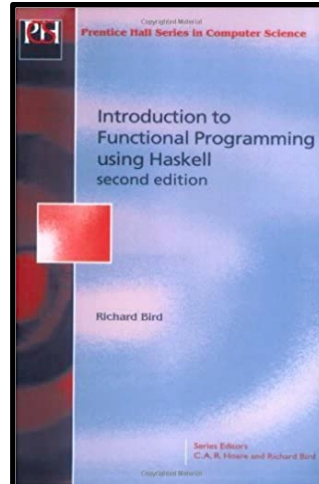
Part 4 - through the work of



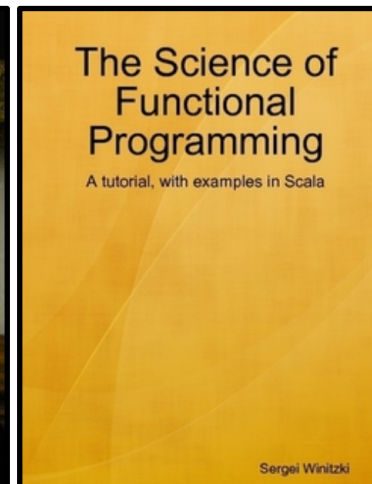
Tony Morris
 @dibblego



<https://presentations.tmorris.net/>



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



Sergei Winitzki
 sergei-winitzki-11a6431

slides by



@philip_schwarz



slideshare <https://www.slideshare.net/pjschwarz>



As promised at the end of **Part 3**, let's now resume looking at **Tony's** talk. We last saw him using **foldr** to implement the **flatten** function. In the next slide he uses **foldr** to implement the **filter** function.



Tony Morris

 @dibblego

right folds replace constructors

Let's filter a list on predicate

Filter is a little bit ugly. Who's never heard of the **filter** function? Everyone has heard of it. They are talking about it outside right now. **Filter**. We are going to take a **predicate** and keep the elements that match that **predicate**. It gets a bit tricky.

filter a list on predicate (p)

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

What are we going to replace **Cons** with? We want to take the **head** at that **Cons**, the **A** or the **B** or the **C** or whatever, and we want to check if it matches a **predicate**. And if it does, we are going to **Cons** that value back on again. If it doesn't, we are just going to leave it off and then move on to the next **Cons** cell.

filter a list on predicate (p)

- let **Cons** = \x -> if p x then **Cons** x else id
- let **Nil** = **Nil**

So we want to say, given **x**, the element that I am visiting, if it matches the **predicate**, then **Cons x**, otherwise do nothing, **id**, the **identity function**. And replace **Nil** with **Nil**.



Tony Morris

 @dibblego

filter a list on predicate (p)

Supposing

```
applyp = if p x then Cons x else id  
list = applyp A (applyp B (applyp C (applyp D Nil)))
```

```
filter p list = foldr (\x -> if p x then Cons x else id) Nil list  
filter p = foldr (\x -> if p x then Cons x else id) Nil  
filter p = foldr (\x -> bool id (Cons x) (p x)) Nil  
filter p = foldr (bool id . Cons <*> p) Nil
```

So, if you ever see this code, you'll know that it is filtering a **list**. Replace **Cons** with `(\x -> if p x then Cons else id)` and **Nil** with **Nil**.

I happen to have a particular aversion to the **if then else** construct by the way. I wish it would die. It takes three arguments for a start. And for those who attended yesterday, what are my favourite functions? How many arguments do those take? One. So I don't like that one.

There is a function that does **if then else** that does take one argument. It is called **bool**. I have used it here. And it does **if then else** with the arguments the right way. There is a right way. It says, if `(p x)`, then `(Cons x)`, else `id`.

That takes one argument. The reason I like that function is because I can get down to that point there (last line above), and that point means I don't have to name so many identifiers. That's just my goal in life. That's not really the point. The point is that I can do that. I can look at this code and I can say: replace **Cons** with `(bool id . Cons <*> p)` and **Nil** with **Nil**. What is it going to do? It is going to filter the **list** on p. OK?

If you found it challenging at all to understand how that last refactoring of the definition of `filter` works, i.e. what the `<*>` operator is doing in the refactored definition, then you are not alone.

```
filter p = foldr (\x -> bool id (Cons x) (p x)) Nil
filter p = foldr (bool id . Cons <*> p) Nil
```

`<*>` is called **apply** or **ap**, but is also known as the **advanced tie fighter** (`|+|` being the **plain tie fighter**), the **angry parent**, and the **sad Pikachu**.

I am familiar with `<*>` as `Applicative`'s **apply** function, which takes a function in a context and the function's argument in that context and returns the result of applying the function to its argument, also in that context.

e.g. if I define function `inc x = x + 1`, then if `inc` is in a `Maybe` context and 3 is also in a `Maybe` context, I can use `<*>` to **apply** `inc` to 3 and return the result 4 in a `Maybe` context:

```
> Just inc <*> Just 3
Just 4
```

Similarly for a `List` context:

```
> [inc] <*> [3]
[4]
```

```
instance Functor ((->) a) where
  fmap = (.)           (a -> b) -> (r -> a) -> r -> b

instance Applicative ((->) a) where
  pure = const         a -> r -> a
  (<*>) f g z = f x (g x) (r -> (a -> b)) -> (r -> a) -> r -> b
  ...
```

Well, it turns out that a function is also an `Applicative`, so if I have a function `f :: a -> b -> c` and a function `g :: a -> b` then I can use `<*>` to create a function `h :: a -> c` which passes its argument `a` to both `f` and `g` as follows: `f a (g a)`.

e.g.

```
> ((+) <*> inc) 3
7
```

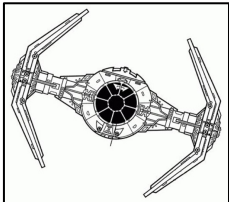
```
<*> :: (r -> (a -> b)) -> (r -> a) -> r -> b
inc  :: Num a => a -> a
(+)  :: Num a => a -> a -> a
```



@philip_schwarz



Tie fighter



advanced Tie fighter

We are using `<*>` to produce the $(\alpha \rightarrow \beta \rightarrow \beta)$ function that we need to pass to `foldr`

`foldr` :: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$

```
foldr      :: (\alpha -> \beta -> \beta) -> \beta -> [\alpha] -> \beta
foldr f e []      = e
foldr f e (x:xs) = f x (foldr f e xs)
```

to be able to implement `filter` as follows

`filter` p = `foldr` (bool id . (:)) `<*>` p []

If we take the signature of `<*>`

`<*>` :: (r -> (a -> b)) -> (r -> a) -> r -> b

`bool` :: a -> a -> Bool -> a

Case analysis for the `Bool` type. `bool x y p` evaluates to `x` when `p` is `False` and evaluates to `y` when `p` is `True`.

This is equivalent to `if p then y else x`; that is, one can think of it as an `if-then-else` construct with its arguments reordered.

and make the following substitutions

a = `Bool`
b = `[a] -> [a]`
r = `a`

```
> inc x = x + 1
> Just inc <*> Just 3
Just 4
> [inc] <*> [3]
[4]
> ((+) <*> inc) 3
7
> ((+) <*> inc) 3 == (+) 3 (inc 3)
True
```

we can understand the signature in our context and see the types that `<*>` is operating on

`<*>` :: (a -> (Bool -> [a] -> [a])) -> (a -> Bool) -> a -> [a] -> [a]

bool id . (:) even f

That's why we are able to refactor

`\x -> bool id ((:) x) (p x)`

```
> :type bool id
bool id :: (a -> a) -> Bool -> a -> a
```

```
> :type (:)
(:) :: a -> [a] -> [a]
```

to

`bool id . (:)` `<*>` p

```
> :type (bool id) . (:)
(bool id) . (:) :: a -> Bool -> [a] -> [a]
```



For what it is worth, the order of the arguments of a conditional **if C then P else Q** does not look so 'wrong' to me if I think that in the **lambda calculus**, **true** is a **binary selector function** returning its first argument, **false** is a **binary selector function** returning its second argument, and the **conditional function** is one that takes a **truth-valued conditional expression** and two other **alternative expressions** and applies the **selector function** that is the result of the first expression to the two other expressions, which results in the **selector function** selecting one of the **alternative expressions** for evaluation.

Conditional Execution in the Lambda Calculus

true = $\lambda x . \lambda y . x$

true 3 4 \rightarrow_{β} 3

false = $\lambda x . \lambda y . y$

false 3 4 \rightarrow_{β} 4

if C then P else Q \rightarrow $\lambda c . \lambda p . \lambda q . cpq$



NOTE: **Lazy evaluation** is required to stop **alternative expressions P** and **Q** from both being evaluated before they are passed to the selector function that is the value of the **conditional** expression.



Tony's talk now turns to a very interesting subject which is a bit advanced and which we are going to cover separately in an upcoming part of this series, so we are now going to skip the next section of his talk and jump to its final part, in which he uses **foldr** to implement a function called **heador** and also recaps on some of the key things that we have learned in his talk.



Tony Morris

 @dibblego

right folds replace constructors

Let's get the head of a list, or a default for no head

```
:: a -> List[a] -> a
```

I find this one quite interesting. So what this function does, is it takes an element **a**, and a list of **as**, and it returns the first element out of the list. The first element. But it could be **Nil**. And if it's **Nil** return the other element that I passed in as an argument. OK, so if this list is **Nil**, return the **a** that is the first argument, otherwise return the first **a** in the list.

What am I going to replace **Cons** with? **const**. **const** the first argument. I agree. And **Nil**? That value there, that's the thing that we are returning if **Nil**, right? So given an entire list, if you ever see **Nil**, return that value, but if you see **Cons**, then return the value that is sitting right at the head of that **Cons**. That will do exactly what our requirements is for this function. So given this list

The head of a list, or default for no head

Supposing

```
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

I want to return **A**, that's the first element of that list. Here is a function that given the **A**, and then the rest of the stuff after the **Cons**, just returns **A**. Replace **Cons**, with that. And replace **Nil**, with that default value that came in as an argument. That will do what it is we need to do.

The head of a list, or default for no head

- let **Cons** = \x _ -> x
- let **Nil** = the default



Tony Morris

 @dibblego

So there is the code.

The head of a list, or default for no head

Supposing

```
constant x _ = x
list = Cons A (Cons B (Cons C (Cons D Nil)))
```

```
heador thedefault list = foldr constant thedefault list
heador thedefault = foldr constant thedefault
heador = foldr constant
```

So who remembers writing **heador** yesterday, in the workshop? We wrote **heador**, right? This is **heador**. And we did it with pattern matching and all of that. We could have just written this: **fold right**, **const**, or **constant**, with that argument.

From <https://hoogle.haskell.org/?hoogle=const>:

const x is a unary function which evaluates to x for all inputs.

```
>>> const 42 "hello"
42
```

```
>>> map (const 42) [0..3]
[42, 42, 42, 42]
```




Tony Morris

 @dibblego

```
$ heador a = foldr const a
```

OK, so, `heador 99 infinity`.

```
$ heador 99 infinity
```

This will not go all the way to the right of that list, because when it gets there, there is just a *Cons* all the way. So I should get 1. And I do:

```
$ heador 99 infinity
1
$
```

Fold right just worked on an **infinite** list. What's the complexity of **heador**? $O(1)$. So, whether or not **fold right** will work on an **infinite list** depends on the **strictness** of the function that we are replacing *Cons* with.

Observations

- `foldr` may work on an **infinite list**.
 - There is no **order** specified, however, there is **associativity**.
 - Depends on the **strictness** of the given function.
 - ?

const, the function I just used, is **lazy**, it ignores the second argument, and therefore it works on an **infinite list**.



Tony Morris

 @dibblego

Observations

- foldr may work on an infinite list.
 - There is no *order* specified, however, there is associativity.
 - Depends on the strictness of the given function.
 - Replaces the *Nil* constructor *if it ever comes to exist*

?

It only replaces *Nil* if *Nil* ever comes to exist. It doesn't exist in **infinity**.

Observations

- foldr may work on an infinite list.
 - There is no *order* specified, however, there is associativity.
 - Depends on the strictness of the given function.
 - Replaces the *Nil* constructor *if it ever comes to exist*
- The expression **foldr Cons Nil** leaves the list unchanged.
 - In other words, passing the list constructors to **foldr** produces an **identity** function

How about this function: **foldr Cons Nil**? Leave the list alone. Replace *Cons* with *Cons* and *Nil* with *Nil*. It does nothing to the list. It's an **identity function**. What's the function that gives me an identity for bool? That's an interesting question? I just showed it to you. I'll let you think about that.



 @philip_schwarz

Tony just explained that doing a **right fold** with *Cons* and *Nil* produces the **identity function**.

In **Parts 1** and **3** we saw that doing a **left fold** with a flipped *Cons* and with *Nil* produces the **reverse** function.

The next slide illustrates the above.

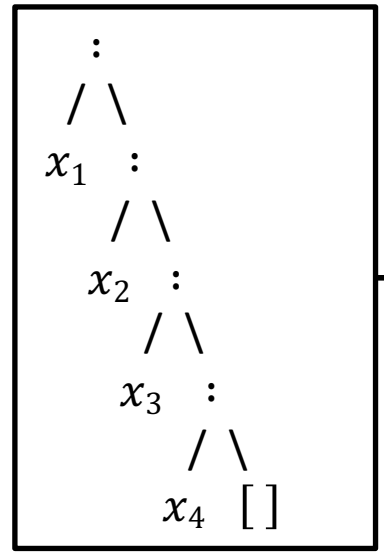
folding a list **right** and **left** using **Cons** and **Nil** results in the **identity** and **reverse** functions

$(:)$ = $\lambda x.\lambda y.x : y$
 (\cdot) = $\lambda x.\lambda y.y : x$
 $(\cdot\cdot)$ = *flip* $(:)$
 where *flip* $f\ x\ y = f\ y\ x$

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

replace:
 $(:)$ with f
 $[]$ with e

$xs = [x_1, x_2, x_3, x_4]$



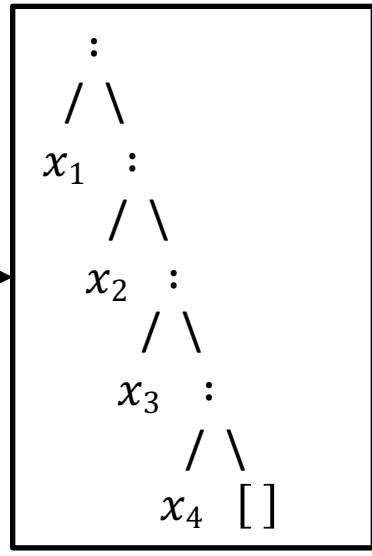
replace:
 $(:)$ with $(:)$
 $[]$ with $[]$

foldr $(:)\ []\ xs$
identity

foldl $(\cdot\cdot)\ []\ xs$
reverse

var $acc = []$
foreach $(x\ \text{in}\ xs)$
 $acc = acc \cdot\cdot x$
return acc

$x_1:(x_2:(x_3:(x_4:[])))$



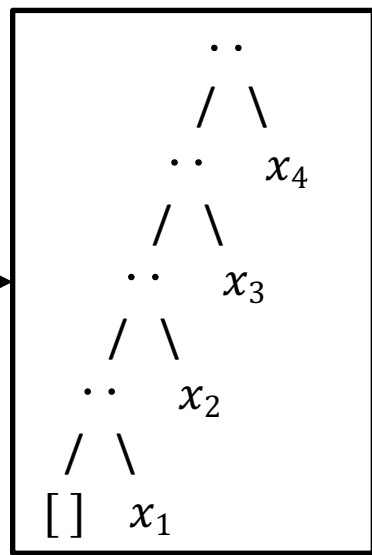
$[x_1, x_2, x_3, x_4]$

foldr $(:)\ [] = \text{identity} = \text{foldr}\ (\lambda x.\lambda y.x : y)\ []$
foldl $(\cdot\cdot)\ [] = \text{reverse} = \text{foldl}\ (\lambda x.\lambda y.y : x)\ []$

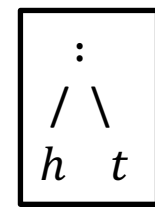
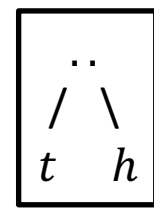
```
> id = foldr (:) []
> rev = foldl (flip (:))
[]
> id [1,2,3,4]
[1,2,3,4]
> rev [1,2,3,4]
[4,3,2,1]
```

```
> id = foldr (\x y -> x:y) []
> rev = foldl (\x y -> y:x) []
> id [1,2,3,4]
[1,2,3,4]
> rev [1,2,3,4]
[4,3,2,1]
```

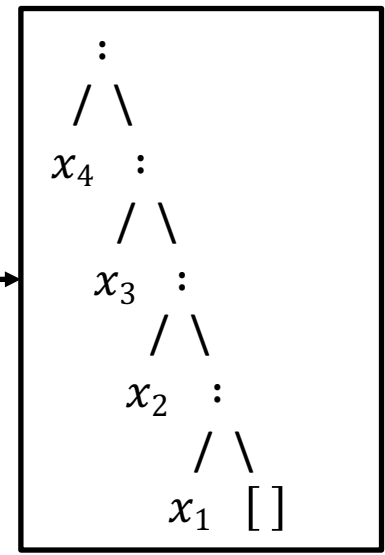
var $acc = []$
foreach $(x\ \text{in}\ xs)$
 $acc = acc \cdot\cdot x$
return acc



$((([] \cdot\cdot x_1) \cdot\cdot x_2) \cdot\cdot x_3) \cdot\cdot x_4$



equivalent to



$x_4 : (x_3 : (x_2 : (x_1 : [])))$

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

var $acc = e$
foreach $(x\ \text{in}\ xs)$
 $acc = f(acc, x)$
return acc

Summary



Tony Morris

 @dibblego

the key intuition

- **left fold** performs a *loop*, just like we are familiar with
- **right fold** performs *constructor replacement*

The key intuition is, the thing to take away is, a **left fold** does a **loop**, and a **right fold** does **constructor replacement**.

If you always remember those two things you'll never go wrong.

from this we derive some observations

- **left fold** will *never* work on an **infinite list**
- **right fold** *may* work on an **infinite list**

Left fold will never work on an **infinite list**. We can see that in the **loop**. **Right fold** might. And these are just independent observations. They have nothing to do with programming languages. I have used **Haskell** as the example. These things are independent of the programming language.

from this we also solve problems

- product
- append
- map
- length
- ...

```
product = foldl (*) 1
append = flip (foldr Cons)
map f = foldr (Cons . f) Nil
length = foldl (const . (+ 1)) 0
```

So we could solve all these problems, as we just did.

Summary



Tony Morris

 @dibblego

I just want to be very clear on this: **fold left** does a **loop**, **fold right** does **constructor replacement**, and there are no footnotes required, this is precise.

- intuitively, this is what list folds do
 - `foldl` performs a **loop**
 - `foldr` performs **constructor replacement**
- this intuition is **precise** and requires no footnotes

I don't want you to ring me up next week and say 'you told me it does **constructor replacement** and it didn't this time'. Yes it did. It always will. OK. There are no footnotes required.

The End

Nil

Thanks!



In the last part of his talk, **Tony** explained that whether or not **fold right** will work on an **infinite list** depends on the **strictness** of the function that we are replacing **Cons** with.

What about in **Scala**? Is there a notion of an **infinite list**? Can a **fold right** function be written which, when used to replace **Cons** with a suitable function, is able to handle an **infinite list**?

In **Part 1** I said that we weren't going to be modelling **infinity** in any of the **Scala** code in this series of slide decks. I have changed my mind about that. I do want to look at how to do **right folds** over large and **infinite lists**. And the technique that we'll be using to do that is also used to address limitations in the **accumulator technique**. So I want to use it for that too.

Which limitations in the **accumulator technique** I hear you ask? Let's cover that in this deck. And since we are there, let's cover a technique that is the dual of the **accumulator trick**, i.e. **tupling**.

And how can an **infinite list** be created in **Scala**? We can use the **iterate** function. Let's cover that in this deck too.

Also, there is something closely related to **folding** that we have not looked at yet, i.e. **scanning**. Let's also cover that in this deck.

So that means the following subjects will be addressed in **Part 5**:

- how to do **right folds** over large and **infinite lists**
- how to get around limitations in the applicability of the **accumulator trick**



In the next four slides we see how **Sergei Winitzki** describes the **iterate** function in his book.

2.3 Converting a single value into a sequence

An **aggregation** converts (“**folds**”) a **sequence** into a **single value**; the **opposite operation** (“**unfolding**”) converts a **single value** into a **sequence**. An example of this task is to **compute the sequence of decimal digits** for a given integer:

```
def digitsOf(x: Int): Seq[Int] = ???
```

```
scala> digitsOf(2405)
res0: Seq[Int] = List(2, 4, 0, 5)
```

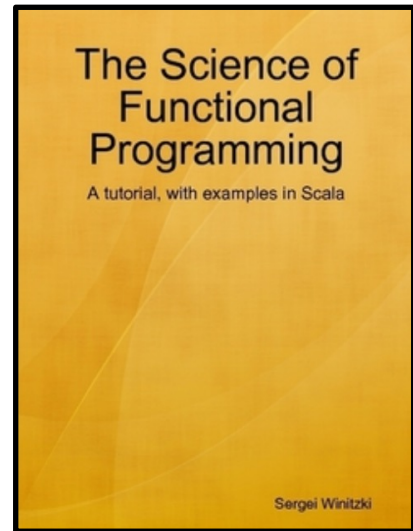
We cannot implement **digitsOf** using **map**, **zip**, or **foldLeft**, because **these methods work only if we already have a sequence**; **but the function digitsOf needs to create a new sequence**. We could create a sequence via the expression `(1 to n)` if the required length of the sequence were known in advance. However, the function **digitsOf** must produce a sequence whose length is determined by a condition that we cannot easily evaluate in advance.

A general “unfolding” operation needs to build a sequence whose length is not determined in advance. This kind of sequence is called a stream. The elements of a stream are computed only when necessary (unlike the elements of List or Array, which are all computed in advance). The unfolding operation will compute the next element on demand; this creates a stream. We can then apply takeWhile to the stream, in order to stop it when a certain condition holds. Finally, if required, the truncated stream may be converted to a list or another type of sequence. In this way, we can generate a sequence of initially unknown length according to any given requirements.

The Scala library has a general stream-producing function Stream.iterate. This function has two arguments, the initial value and a function that computes the next value from the previous one:

```
scala> Stream.iterate(2) { x => x + 10 }
res0: Stream[Int] = Stream(2, ?)
```

The stream is ready to start computing the next elements of the **sequence** (so far, only the first element, 2, has been computed).



Sergei Winitzki

In order to see the next elements, we need to stop the **stream** at a finite size and then convert the result to a list:

```
scala> Stream.iterate(2) { x => x + 10 }.take(6).toList
res1: List[Int] = List(2, 12, 22, 32, 42, 52)
```

If we try to evaluate **toList** on a **stream** without first limiting its size via **take** or **takeWhile**, the program will keep producing more elements of the **stream** until it runs out of memory and crashes.

Streams are similar to **sequences**, and methods such as **map**, **filter**, and **flatMap** are also defined for **streams**. For instance, the method **drop** skips a given number of initial elements:

```
scala> Seq(10, 20, 30, 40, 50).drop(3)
res2: Seq[Int] = List(40, 50)
```

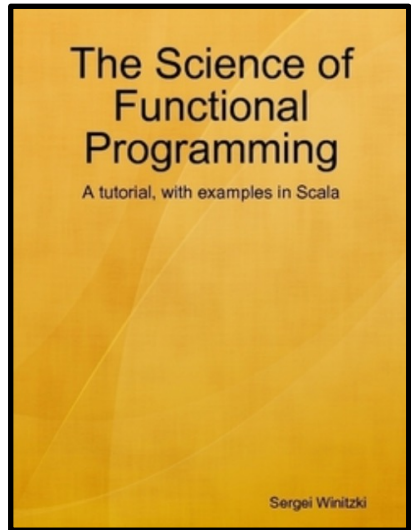
```
scala> Stream.iterate(2) { x => x + 10 }.drop(3)
res3: Stream[Int] = Stream(32, ?)
```

This example shows that in order to evaluate **drop(3)**, the **stream** had to compute its elements up to 32 (but **the subsequent elements are still not computed**).

To figure out the code for **digitsOf**, we first write this function as a **mathematical formula**. To compute the digits for, say, **n = 2405**, we need to divide **n** repeatedly by **10**, getting a sequence **n_k** of intermediate numbers (**n₀ = 2405**, **n₁ = 240**, ...) and the corresponding sequence of last digits, **n_k mod 10** (in this example: **5**, **0**, ...). The sequence **n_k** is defined using **mathematical induction**:

- **Base case:** **n₀ = n**, where **n** is the given initial integer.
- **Inductive step:** **n_{k+1} = $\lfloor \frac{n_k}{10} \rfloor$** for **k = 1, 2, ...**

Here $\lfloor \frac{n_k}{10} \rfloor$ is the mathematical notation for the integer division by 10.



Sergei Winitzki

Let us tabulate the evaluation of the sequence n_k for $n = 2405$:

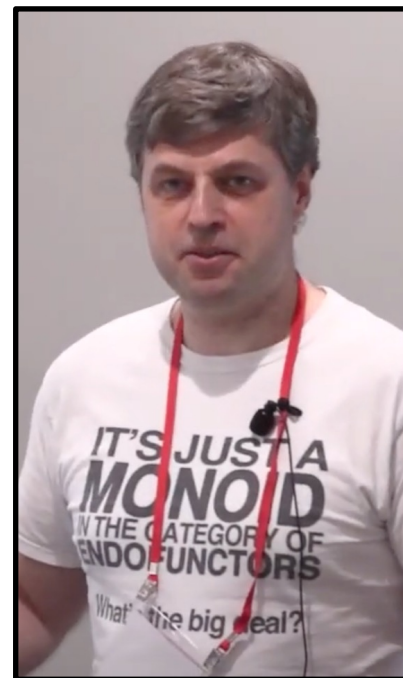
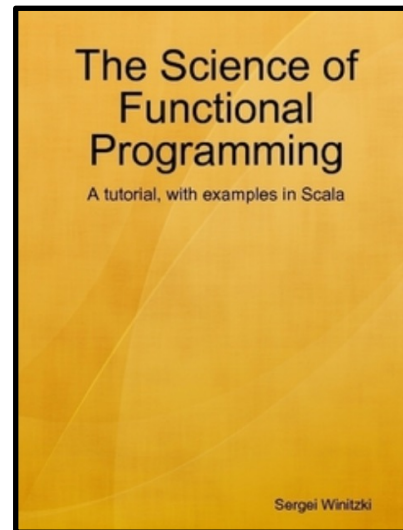
$k =$	0	1	2	3	4	5	6
$n_k =$	2405	240	24	2	0	0	0
$n_k \bmod 10 =$	5	0	4	2	0	0	0

The numbers n_k will remain all zeros after $k = 4$. It is clear that the useful part of the **sequence** is before it becomes all zeros. In this example, the sequence n_k needs to be stopped at $k = 4$. The **sequence** of digits then becomes $[5, 0, 4, 2]$, and we need to reverse it to obtain $[2, 4, 0, 5]$. For reversing a **sequence**, the **Scala** library has the standard method **reverse**. So, a complete implementation for **digitsOf** is:

```
def digitsOf(n: Int): Seq[Int] =
  if (n == 0) Seq(0) else { // n == 0 is a special case.
    Stream.iterate(n) { nk => nk / 10 }
      .takeWhile { nk => nk != 0 }
      .map { nk => nk % 10 }
      .toList.reverse
  }
```

We can shorten the code by using the syntax such as $(_ \% 10)$ instead of $\{ nk => nk \% 10 \}$,

```
def digitsOf(n: Int): Seq[Int] =
  if (n == 0) Seq(0) else { // n == 0 is a special case.
    Stream.iterate(n) ( _ / 10 )
      .takeWhile ( _ != 0 )
      .map ( _ % 10 )
      .toList.reverse
  }
```




Sergei Winitzki

The type signature of the method `Stream.iterate` can be written as

```
def iterate[A](init: A)(next: A => A): Stream[A]
```

and shows a close correspondence to a definition by mathematical induction. The base case is the first value, `init`, and the inductive step is a function, `next`, that computes the next element from the previous one. It is a general way of creating sequences whose length is not determined in advance.



I want to show you another example of using `iterate`. There is one example that I find quite interesting, which is the computation of `fibonacci numbers` using a technique which is called `tupling` and which is the dual of the `accumulator technique` that we have already seen.

Before we can look at that example, we are going to do the following:

- The next two slides are from **Part 2** and remind us of how the `accumulator technique` can sometimes be used to transform a program so that it becomes `tail recursive`. Just skip the slides if they are still fresh in your mind.
- The two slides after that, remind us of how the `accumulator technique` can sometimes be used to improve the `efficiency` of a program.
- The subsequent slide briefly explains how the `tupling technique` can sometimes be used to increase the `efficiency` of a program.

The Science of Functional Programming

A tutorial, with examples in Scala

Sergei Winitzki

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

```
lengthS(Seq(1, 2, ..., 100000))  
= 1 + lengthS(Seq(2, ..., 100000))  
= 1 + (1 + lengthS(Seq(3, ..., 100000)))  
= ...
```

The function body of `lengthS` will evaluate the **inductive step**, that is, the “else” part of the “if/else”, about **100_000** times. Each time, the sub-expression with nested computations $1+(1+(\dots))$ will get larger.

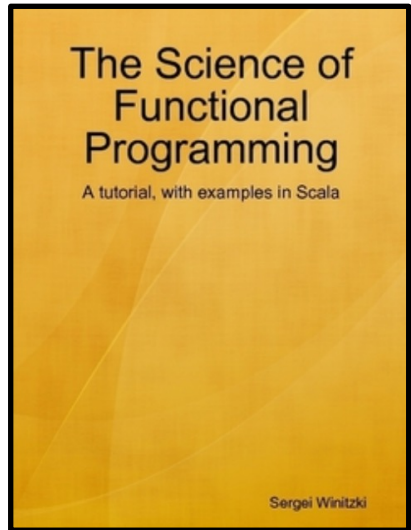
This intermediate sub-expression needs to be held somewhere in memory, until at some point the function body goes into the **base case** and returns a value. When that happens, the entire intermediate sub-expression will contain about **100_000 nested function calls still waiting to be evaluated**.

This sub-expression is held in a special area of memory called **stack memory**, where the not-yet-evaluated **nested function calls** are held in the order of their calls, as if on a “**stack**”. 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.

A 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:

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

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(...)`. In the code of `lengthT`, **recursive calls** never occur within any sub-expressions.



Sergei Winitzki

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

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 has a feature for checking automatically 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.

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

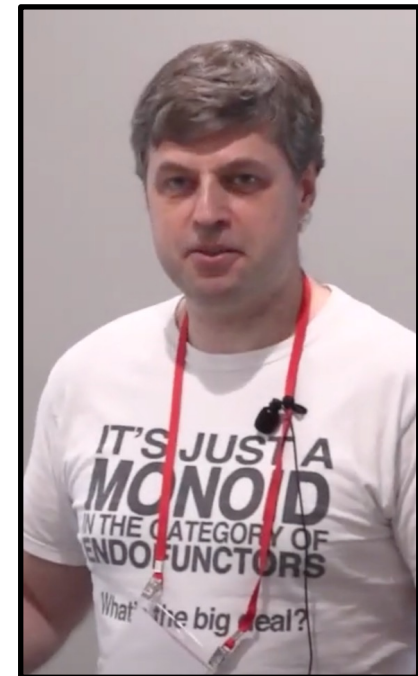
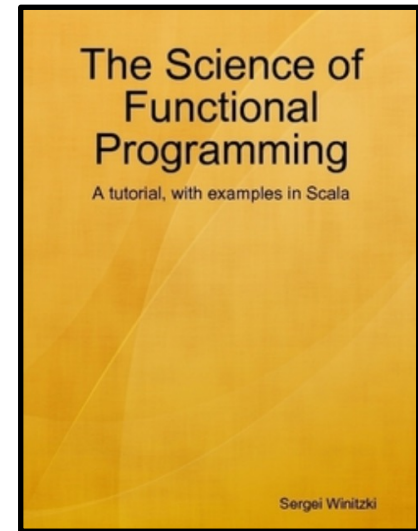
```
def lengthS(s: Seq[Int]): Int =  
  if (s.isEmpty) 0  
  else 1 + lengthS(s.tail)
```

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

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

All sub-expressions such as `1 + 1` and `1 + 2` 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** to obtain 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** (`1 + res`) 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



 @philip_schwarz

That was a reminder of how the **accumulator technique** can sometimes be used to transform a program so that it becomes **tail recursive**.

The next two slides remind us of how the **accumulator technique** can sometimes be used to improve the **efficiency** of a program.



If we rewrite a **recursive function** using the **accumulator technique** then we end up with a **tail recursive function**, i.e. one that is **stack-safe**. In **Part 3** we saw that sometimes the version using the **accumulator** is more **efficient** than the version that doesn't, but at other times it can be less **efficient**, or even just as **efficient**.

Here is how **Richard Bird** expresses the case where it is more **efficient**: "By adding an extra parameter to a function we can sometimes improve the running time. The most important use of this idea is to eliminate possibly expensive **+** operations from a program"

$$T(\text{reverse})(n) = \Theta(n^2)$$

```
reverse      :: [\alpha] -> [\alpha]
reverse []   = []
reverse (x : xs) = reverse xs # [x]
```

$$T(\text{reverse}')(n) = \Theta(n)$$

```
reverse'     :: [\alpha] -> [\alpha]
reverse' xs  = accum [] xs

accum ws []  = ws
accum ws (x : xs) = accum (x : ws) xs
```

the **accumulator** version is **more efficient**

$$T(\text{concat})(m, n) = \Theta(mn)$$

```
concat      :: [[\alpha]] -> [\alpha]
concat []   = []
concat (xs : xss) = xs # concat xss
```

$$T(\text{concat}')(m, n) = \Theta(m^2n)$$

```
concat'     :: [[\alpha]] -> [\alpha]
concat' xss = accum [] xss

accum ws []  = ws
accum ws (xs : xss) = accum (ws # xs) xss
```

the **accumulator** version is **less efficient**

= *append*

$$T(\text{append})(m, n) = \Theta(m)$$

```
append     :: [\alpha] -> [\alpha] -> [\alpha]
append []  = ys
append (x : xs) ys = x : (append xs ys)
```

$$T(\text{append}')(m, n) = \Theta(m)$$

```
append'    :: [\alpha] -> [\alpha] -> [\alpha]
append' xs ys = accum ys (reverse xs)

accum ys []  = ys
accum ys (x : xs) = accum (x : ys) xs
```

the **accumulator** version is **just as efficient**



Since we saw, also in **Part 3**, that eliminating **foldr** leads to a **recursive** definition of a function and eliminating **foldl** leads to a **tail-recursive** definition of the function, one that uses an **accumulator**, this slide is simply a restatement of the previous one in which the **recursive** definition has been replaced by a definition using **foldr** and the **tail-recursive** definition (using an **accumulator**) has been replaced by a definition using **foldl**.

$$T(\text{reverse})(n) = \Theta(n^2)$$

```
reverse  ::  [α] → [α]
reverse  =  foldr snoc []
           where snoc x xs = append xs [x]
```

$$T(\text{reverse}')(n) = \Theta(n)$$

```
reverse' ::  [α] → [α]
reverse'  =  foldl prefix []
           where prefix xs x = x : xs
```

the *foldl* version
is **more efficient**

$$T(\text{concat})(m, n) = \Theta(mn)$$

```
concat  ::  [[α]] → [α]
concat  =  foldr append []
```

$$T(\text{concat}')(m, n) = \Theta(m^2n)$$

```
concat' ::  [[α]] → [α]
concat'  =  foldl append []
```

the *foldl* version
is **less efficient**

$$T(\text{append})(m, n) = \Theta(m)$$

```
append  ::  [α] → [α] → [α]
append xs ys = foldr (:) ys xs
```

$$T(\text{append}')(m, n) = \Theta(m)$$

```
append' ::  [α] → [α] → [α]
append' xs ys = foldl scon ys (reverse' xs)
               where scon xs x = x : xs
```

the *foldl* version
is **just as efficient**



That was a reminder of how the **accumulator technique** can sometimes be used to improve the **efficiency** of a program.

The next slide briefly explains how the **tupling technique** can sometimes be used to increase the **efficiency** of a program.

7.4 Tupling

The technique of program optimisation known as **tupling** is dual to that of **accumulating parameters**: **a function is generalised, not by including an extra argument, but by including an extra result**. Our aim in this section is to illustrate this **important technique** through a number of instructive examples.

...

7.4.2 Fibonacci function

Another example where **tupling** can improve the **order of growth** of the **time complexity** of a program is provided by the **Fibonacci** function.

$$\begin{aligned} \text{fib } 0 &= 0 \\ \text{fib } 1 &= 1 \\ \text{fib } (n + 2) &= \text{fib } n + \text{fib } (n + 1) \end{aligned}$$

The time to evaluate $\text{fib } n$ by these equations is given by $T(\text{fib})(n)$, where

$$\begin{aligned} T(\text{fib})(0) &= O(1) \\ T(\text{fib})(1) &= O(1) \\ T(\text{fib})(n + 2) &= T(\text{fib})(n) + T(\text{fib})(n + 1) + O(1) \end{aligned}$$

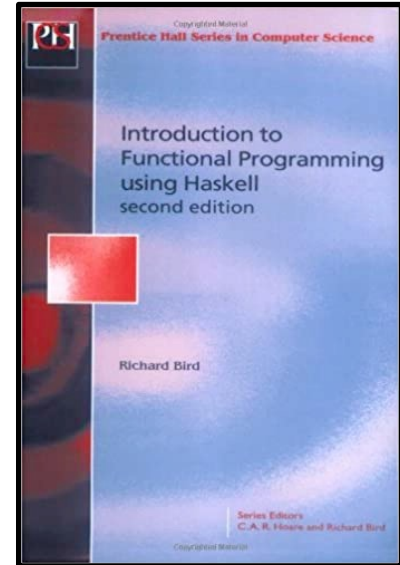
The timing function $T(\text{fib})$ therefore satisfies equations very like that of fib itself. It is easy to check by induction that $T(\text{fib})(n) = \Theta(\text{fib } n)$, so the time to compute fib is proportional to the size of the result. Since $\text{fib}(n) = \Theta(\phi^n)$, where ϕ is the **golden ratio** $\phi = (1 + \sqrt{5})/2$, the time is therefore **exponential** in n . Now consider the function fibtwo defined by

$$\text{fibtwo } n = (\text{fib } n, \text{fib } (n + 1))$$

Clearly, $\text{fib } n = \text{fst}(\text{fibtwo } n)$. Synthesis of a **recursive** program for fibtwo yields

$$\begin{aligned} \text{fibtwo } 0 &= (0, 1) \\ \text{fibtwo } (n + 1) &= (b, a + b), \text{ where } (a, b) = \text{fibtwo } n \end{aligned}$$

It is clear that this program takes **linear time**. **In this example the tupling strategy leads to a dramatic increase in efficiency, from exponential to linear.**



Richard Bird



We have actually already been introduced (in **Part 1**) to the idea of computing **Fibonacci** numbers more efficiently by using **tupling**, but in a less formal way. See below for a reminder.

 @philip_schwarz



```

fib           :: Nat → Nat
fib Zero     = Zero
fib (Succ Zero) = Succ Zero
fib (Succ (Succ n)) = fib (Succ n) + fib n
  
```

introduce tupling

```

fib :: Nat → Nat
fib = fst · foldn g (Zero, Succ Zero)
  where g(m,n) = (n, m + n)
  
```

```

foldn           :: (α → α) → α → Nat → α
foldn h c Zero = c
foldn h c (Succ n) = h (foldn h c n)
  
```



```

val fib: Nat => Nat = {
  case Zero      => Zero
  case Succ(Zero) => Succ(Zero)
  case Succ(Succ(n)) => fib(Succ(n)) + fib(n)
}
  
```

introduce tupling

```

def fib(n: Nat): Nat = {

  def fst(pair: (Nat, Nat)): Nat =
    pair match { case (n,_) => n }

  def g(pair: (Nat, Nat)): (Nat, Nat) =
    pair match { case (m,n) => (n, m + n) }

  fst( foldn(g, (Zero, Succ(Zero)), n) )
}
  
```

```

def foldn[A](h: A => A, c: A, n: Nat): A =
  n match {
    case Zero => c
    case Succ(n) => h(foldn(h,c,n))
  }
  
```

On the previous slide we saw a definition of the **fibonacci** function that uses **tupling** in conjunction with **foldn**, which operates on the **Nat** type, i.e. **Zero**, **Succ(Zero)**, **Succ(Succ(Zero))**, etc.

What if we want to use **Int** rather than **Nat**? Just for curiosity: does it make any sense to try and use **foldLeft** to write a definition of the **fibonacci** function that uses **tupling**?

It does make some sense. Here is how we can do it:

```
def fibFoldLeft(n: Int): Long =  
  (1 to n) foldLeft ((0L, 1L)){  
    case ((f1, f2), _) => (f2, f1 + f2)  
  }._1
```

The odd thing about the above function is that while it creates a **sequence** 1 to n, it uses the sequence purely to determine how many times to **iterate**: it doesn't actually use the values in the **sequence** – e.g. it would work just as well if the **sequence** were 101 to 100 + n rather than 1 to n.

Still, it works. Let's compare the time taken to compute a large-ish **Fibonacci** number using both **fibFoldLeft** and the traditional definition of the **Fibonacci** function:

```
def fibonacci(n: Int): Long = n match {  
  case 0 => 0L  
  case 1 => 1L  
  case n =>  
    fibonacci(n - 1) + fibonacci(n - 2)  
}
```



```
def eval[A](expression: => A): (A, Duration) = {
  def getTime = System.currentTimeMillis()
  val startTime = getTime
  val result = expression
  val endTime = getTime
  val duration = endTime - startTime
  (result, Duration(duration, "ms"))
}
```



@philip_schwarz

Here on the left is an **eval** function that we can use to time the execution of different definitions of the **fibonacci** function.

Computing **fibonacci** of **50** with the standard **recursive** definition takes a stonking **82 seconds!!!**.

```
def fibonacci(n: Int): Long = n match {
  case 0 => 0L
  case 1 => 1L
  case n =>
    fibonacci(n - 1) + fibonacci(n - 2)
}
```

Computing **fibonacci** of **10,000** fails due to a **stack overflow** error.

```
scala> val (result, duration) = eval(fibonacci(50))
val result: Long = 12586269025
val duration: Duration = 82000 milliseconds
```

Here we change the return type to **BigDecimal** to allow for very large results.

```
scala> fibonacci(10_000)
java.lang.StackOverflowError
  at fibonacci(<console>:1)
  at fibonacci(<console>:4)
  ...
```

Computing **fibonacci** of **50** with **fibFoldLeft** takes only **1 millisecond!!!**

```
def fibFoldLeft(n: Int): BigDecimal =
  (1 to n).foldLeft((BigDecimal(0), BigDecimal(1))) {
    case ((f1, f2), _) => (f2, f1 + f2)
  }._1
```

Computing **fibonacci** of **10,000** takes only **8 milliseconds!!!**

```
scala> val (result, duration) =
eval(fibFoldLeft(50))
val result: BigDecimal = 12586269025
val duration: Duration = 1 millisecond
```

```
scala> val (result, duration) = eval(fibFoldLeft(10_000))
val result: BigDecimal = 3.364476487643178326662161200510745E+2089
val duration: Duration = 8 milliseconds
```



Eleven slides ago I said I wanted to show you a second example of using the **iterate** function. Now that we have gone through the following we are finally in a position to look at that example:

- a refresher of the **accumulator technique**
- a brief introduction to the **tupling technique**
- a look at how to implement a **Fibonacci** function using **tupling**

The example consists of implementing the **Fibonacci** function using **iterate**. We are going to use the same **tupling** technique that we have just used in **fibFoldLeft**. When **Sergei Winitzki** introduced the **iterate** function, we saw that it is a function that returns a **Stream**, e.g. **Stream.iterate(2) { x => x + 10 }**. Since **Scala 2.13** however, the **Stream** class is deprecated in favour of **LazyList**, so we are going to use **LazyList.iterate**, whose signature is otherwise identical to that of **Stream.iterate**.

The **fibIterate** function below is similar to **fibFoldLeft** from the previous slide in that it uses the same **tupling** approach, but it is simpler because it doesn't have to create a sequence whose data it doesn't even need. It also performs the same way.

```
/** An infinite LazyList that repeatedly applies a given function
 *  to a start value.
 *
 *  @param start the start value of the LazyList
 *  @param f     the function that's repeatedly applied
 *  @return     the Stream returning the infinite sequence
 *             of values `start, f(start), f(f(start)), ...`
 */
def iterate[A](start: A)(f: A => A): LazyList[A] = ...
```

<https://www.scala-lang.org/blog/2018/06/13/scala-213-collections.html>

LazyList Is Preferred Over Stream

Stream is deprecated in favor of **LazyList**. As its name suggests, a **LazyList** is a linked list whose elements are lazily evaluated. An important semantic difference with **Stream** is that in **LazyList** both the head and the tail are lazy, whereas in **Stream** only the tail is lazy.

```
def fibIterate(n: Int) : BigDecimal =
  LazyList.iterate ((BigDecimal(0), BigDecimal(1))){
    case (f1, f2) => (f2, f1 + f2)
  }(n)._1
```

```
scala> val (result, duration) = eval(fibIterate(50))
val result: BigDecimal = 12586269025
val duration: Duration = 1 millisecond
```

fibIterate is as fast as **fibFoldLeft**.



Instead of using **LazyList**'s **take** function, we get the n^{th} element of the **LazyList**, which is a tuple, and take the 1st element of the latter.

```
scala> val (result, duration) = eval(fibIterate(10_000))
val result: BigDecimal = 3.364476487643178326662161200510745E+2089
val duration: Duration = 7 milliseconds
```



Remember how the naive **Scala** definition of the **fibonacci** function was taking over 80 seconds to compute the 50th **Fibonacci** number? Well the **Haskell** version seems to be even slower!

```
fib 0 = 0
fib 1 = 0
fib n = fib (n-1) + fib (n-2)
```

```
> fib 10
55
> fib 20
6765
> fib 30
832040
> fib 40
I got bored of waiting
so I interrupted it
```



Here is the **Haskell** version **fibIterate**. Just like the **Scala** version, it is blazingly fast.

```
fibIterate n = fst (fib2 !! n)
  where fib2 = iterate f (0,1)
        where f (a,b) = (b,a+b)
```

```
> fibIterate 50
12586269025
> fibIterate 10000
336447648764317832666216120051075433103021484606800639065647699746800814421666623681555955136337340255820653326808361593737
347904838652682630408924630564318873545443695598274916066020998841839338646527313000888302692356736131351175792974378544137
521305205043477016022647583189065278908551543661595829872796829875106312005754287834532155151038708182989697916131278562650
331954871402142875326981879620469360978799003509623022910263681314931952756302278376284415403605844025721143349611800230912
082870460889239623288354615057765832712525460935911282039252853934346209042452489294039017062338889910858410651831733604374
707379085526317643257339937128719375877468974799263058370657428301616374089691784263786242128352581128205163702980893320999
057079200643674262023897831114700540749984592503606335609338838319233867830561364353518921332797329081337326426526339897639
227234078829281779535805709936910491754708089318410561463223382174656373212482263830921032977016480547262438423748624114530
938122065649140327510866433945175121615265453613331113140424368548051067658434935238369596534280717687753283482343455573667
197313927462736291082106792807847180353291311767789246590899386354593278945237776744061922403376386740040213303432974969020
283281459334188268176838930720036347956231171031012919531697946076327375892535307725523759437884345040677155557790564504430
166401194625809722167297586150269684431469520346149322911059706762432685159928347098912847067408620085871350162603120719031
720860940812983215810772820763531866246112782455372085323653057759564300725177443150515396009051686032203491632226408852488
524331580515348496224348482993809050704834824493274537326245677558790891871908036620580095947431500524025327097469953187707
243768259074199396322659841474981936092852239450397071654431564213281576889080587831834049174345562705202235648464951961124
602683139709750693826487066132645076650746115126775227486215986425307112984411826226610571635150692600298617049454250474913
78115154139941550671256271197133252763631939606902895650288268608362241082050562430701794976171121233066073310059947366875
>
```



Next, let's look at some of the things that **Richard Bird** has to say about the **iterate** function.

 @philip_schwarz

...there are three kinds of **list**:

- A **finite list**, which is built from `(:)` and `[]`; for example, `1:2:3:[]`
- A **partial list**, which is built from `(:)` and **undefined**; for example, the list `filter (<4) [1..]` is the **partial** list `1:2:3:undefined`. We know there is no integer after 3 that is less than 4, but **Haskell** is an evaluator, not a theorem prover, so it ploughs away without success looking for more answers.
- An **infinite list**, which is built from `(:)` alone; for example, `[1..]` is the **infinite** list of nonnegative integers.

All three kinds of list arise in everyday programming. Chapter 9 is devoted to exploring the world of **infinite lists** and their uses. For example, the prelude function `iterate` returns an **infinite list**:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

In particular, `iterate (+1) 1` is an **infinite** list of the positive integers, a value we can also write as `[1..]`.

The type of `1 `div` 0` is an integral number.

```
ghci> :type 1 `div` 0
1 `div` 0 :: Integral a => a
```

The expression `1 `div` 0` is therefore well-formed and possesses a value.

```
ghci> 1 `div` 0
*** Exception: divide by zero
```

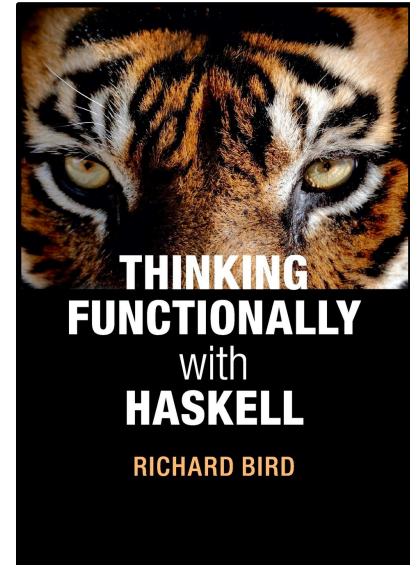
GHCi returns an error message. So what is the type of `1 `div` 0`? The answer is that it is a special value, written mathematically as \perp and pronounced '**bottom**'.

In fact, **Haskell** provides a predefined name for this value, except that it is called **undefined**, not \perp .

```
ghci> :type undefined
undefined :: a

ghci> undefined
*** Exception: Prelude.undefined
```

Haskell is not expected to produce the value \perp . It may return with an error message, or remain perpetually silent, computing an **infinite loop**, until we interrupt the computation. It may even cause GHCi to crash.



Richard Bird

...consider the following three definitions of the standard prelude function *iterate*:

iterate1 $f\ x = x : \textit{iterate1}\ f\ (f\ x)$

iterate2 $f\ x = xs$ where $xs = x : \textit{map}\ f\ xs$

iterate3 $f\ x = x : \textit{map}\ f\ (\textit{iterate3}\ f\ x)$

All three functions have type $(a \rightarrow a) \rightarrow a \rightarrow [a]$ and produce an **infinite list** of **iterates** of f applied to x . The three functions are equal... The first definition is the one used in the standard prelude, but it does not create a **cyclic** list. The second definition does, and the third is obtained from the second by eliminating the *where* clause.

Assuming $f\ x$ can be computed in constant time, the first definition takes $\theta(n)$ steps to compute the first n elements of the result, but the third takes $\theta(n^2)$ steps:

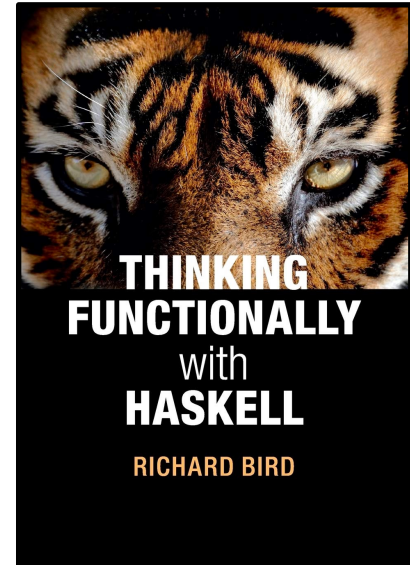
```
iterate3 (2 *) 1
= 1 : map (2 *) (iterate3 (2 *) 1)
= 1 : 2 : map (2 *) (map (2 *) (iterate3 (2 *) 1))
= 1 : 2 : 4 : map (2 *) (map (2 *) (map (2 *) (iterate3 (2 *) 1)))
```

Evaluating the n^{th} element requires n applications of $(2 *)$, so it takes $\theta(n^2)$ to produce the first n elements.

That leaves the second definition. Does it take **linear** or **quadratic** time? The evaluation of *iterate2* $(2 *)\ 1$ proceeds as follows:

```
 $xs$       where  $xs = 1 : \textit{map}\ (2 *)\ xs$ 
 $1:ys$    where  $ys = \textit{map}\ (2 *)\ (1:ys)$ 
 $1:2:zs$  where  $zs = \textit{map}\ (2 *)\ (2:zs)$ 
 $1:2:4:ts$  where  $ts = \textit{map}\ (2 *)\ (4:zs)$ 
```

Each element of the result is produced in constant time, so *iterate2* $(2 *)\ 1$ takes $\theta(n)$ steps to produce n elements.



Richard Bird



Now that we have covered the **iterate** function and the **tupling** technique, let's turn to the subject of **scanning**.

In the next two slides we see how **Sergei Winitzki** describes **scanning** in his book.

2.4 Transforming a sequence into another sequence

We have seen methods such as `map` and `zip` that transform sequences into sequences. However, these methods cannot express a general transformation where the elements of the new sequence are defined by induction and depend on previous elements.

An example of this kind is computing the **partial sums** of a given **sequence** x_i , say $b_k = \sum_{i=0}^{k-1} x_i$. This formula defines $b_0 = 0$, $b_1 = x_0$, $b_2 = x_0 + x_1$, $b_3 = x_0 + x_1 + x_2$, etc. A definition via **mathematical induction** may be written like this:

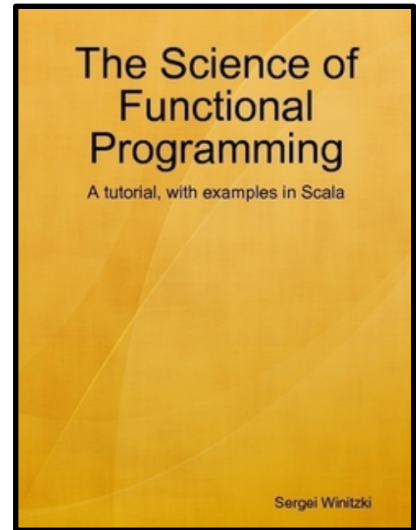
- **Base case:** $b_0 = 0$
- **Inductive step:** Given b_k , we define $b_{k+1} = b_k + x_k$ for $k = 0, 1, 2, \dots$

The **Scala** library method `scanLeft` implements a general, **sequence** to **sequence** transformation defined in this way. The code implementing the partial sums is

```
def partialSums(xs: Seq[Int]): Seq[Int] =  
  xs.scanLeft(0){ (x, y) => x + y }  
  
scala> partialSums(Seq(1, 2, 3, 4))  
val res0: Seq[Int] = List(0, 1, 3, 6, 10)
```

The first argument of `scanLeft` is the **base case**, and the second argument is an **updater function** describing the **inductive step**. In general, the type of elements of the **second sequence** is different from that of the **first sequence**. The updater function takes an element of the first sequence and a previous element of the second sequence, and returns the next element of the second sequence. Note that the result of `scanLeft` is one element longer than the original **sequence**, because the **base case** provides an **initial value**.

Until now, we have seen that `foldLeft` is sufficient to re-implement almost every method that works on sequences, such as `map`, `filter`, or `flatten`. Let us show, as an illustration, how to implement the method `scanLeft` via `foldLeft`.

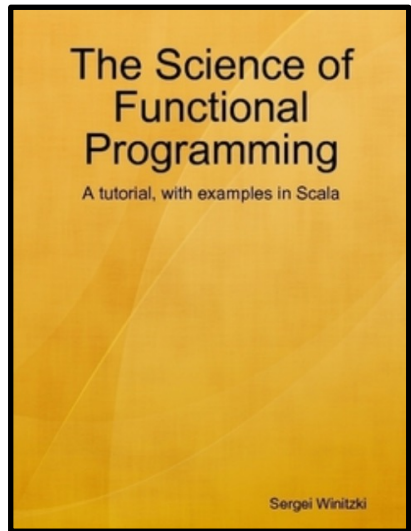


Sergei Winitzki

In the implementation, the **accumulator** contains the **previous element** of the **second sequence** together with a **growing fragment** of that **sequence**, which is updated as we **iterate** over the **first sequence**. The code is

```
def scanLeft[A, B](xs: Seq[A])(b0: B)(next: (B, A) => B): Seq[B] = {
  val init: (B, Seq[B]) = (b0, Seq(b0))
  val (_, result) = xs.foldLeft(init) {
    case ((b, seq), x) =>
      val newB = next(b, x)
      (newB, seq :+ newB)
  }
  result
}
```

To implement the (nameless) **updater function** for **foldLeft** we used the **Scala** feature that makes it easier to define functions with several arguments containing tuples. In our case, the **updater function** in **foldLeft** has two arguments: the first is a tuple $(B, Seq[B])$, the second is a value of type A . The pattern expression `case ((b, seq), x) =>` appears to match a nested tuple. In reality, this expression matches the two arguments of the **updater function** and, at the same time, deconstructs the tuple argument as (b, seq) .



Sergei Winitzki




 @philip_schwarz

While in his explanation of the **scanLeft** function, **Sergei** showed us (for illustration only) that it is possible to implement **scanLeft** using **foldLeft**, I was a bit surprised by the fact that he did not mention that the meaning of the **scanLeft** function is very closely related to that of **foldLeft**.

That close relationship is one of the things we are going to learn in the next five slides, in which **Richard Bird** explains left and right **scans**.


Before we do that, I am just going to have a go at a **Haskell** version of **Sergei's** implementation of **scanLeft** in terms of **foldLeft** (again, for illustration only).

```
def scanLeft[A, B](xs: Seq[A])(b0: B)(next: (B, A) => B): Seq[B] = {
  val init: (B, Seq[B]) = (b0, Seq(b0))
  val (_, result) = xs.foldLeft(init) {
    case ((b, seq), x) =>
      val newB = next(b, x)
      (newB, seq :+ newB)
  }
  result
}
```



```
scala> scanleft(List(1,2,3,4))(0)(_+_ )
val res0: Seq[Int] = List(0, 1, 3, 6, 10)
scala>
```

```
scanleft f e xs = snd (foldl g (e, [e]) xs)
  where g (e, res) x = (next, res ++ [next])
          where next = f e x
```



```
Haskell> scanleft (+) 0
[1,2,3,4]
[0,1,3,6,10]
```

4.5.2 Scan left

Sometimes it is convenient to apply a *foldl* operation to every initial segment of a list. This is done by a function *scanl* pronounced '**scan left**'. For example,

$$\text{scanl } (\oplus) e [x_0, x_1, x_2] = [e, e \oplus x_0, (e \oplus x_0) \oplus x_1, ((e \oplus x_0) \oplus x_1) \oplus x_2]$$

In particular, *scanl* (+) 0 computes the list of accumulated sums of a list of numbers, and *scanl* (×) 1 [1..n] computes a list of the first *n* **factorial** numbers. ... We will give two programs for *scanl*; the first is the **clearest**, while the second is **more efficient**. For the first program we will need the function *inits* that returns the list of all **initial segments** of a list. For Example,

$$\text{inits } [x_0, x_1, x_2] = [[], [x_0], [x_0, x_1], [x_0, x_1, x_2]]$$

The **empty list** has only one **segment**, namely the **empty list** itself; A list (*x:xs*) has the **empty list** as its shortest **initial segment**, and all the other **initial segments** begin with *x* and are followed by an **initial segment** of *xs*. Hence

$$\begin{aligned} \text{inits} &:: [\alpha] \rightarrow [[\alpha]] \\ \text{inits } [] &= [[]] \\ \text{inits } (x:xs) &= [] : \text{map } (x:) (\text{inits } xs) \end{aligned}$$

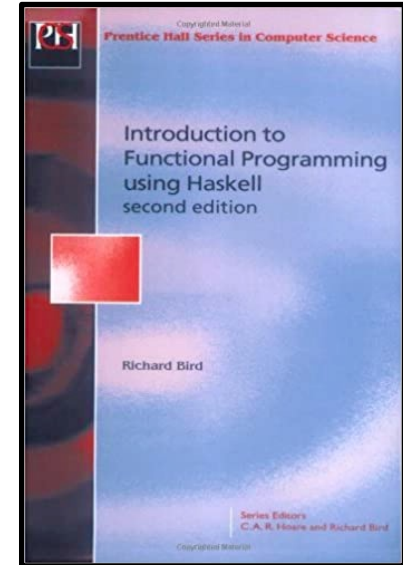
The function *inits* can be defined more succinctly as an instance of *foldr* :

$$\text{inits} = \text{foldr } f [[]] \text{ where } f x xss = [] : \text{map } (x:) xss$$

Now we define

$$\begin{aligned} \text{scanl} &:: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta] \\ \text{scanl } f e &= \text{map } (\text{foldl } f e) . \text{inits} \end{aligned}$$

This is the clearest definition of *scanl* but it leads to an **inefficient program**. The function *f* is applied *k* times in the evaluation of



Richard Bird



By the way, just for completeness, the appendix of **Richard Bird's** book, contains the following additional definition of *scanl*

$$\begin{aligned} \text{scanl} & \quad \text{:: } (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta] \\ \text{scanl } f \ e \ xs = e & : \text{scanl}' \ f \ e \ xs \\ \text{where } \text{scanl}' \ f \ e \ [] & \quad = [] \\ \text{scanl}' \ f \ e \ (y:ys) & = \text{scanl } f \ (f \ e \ y) \ ys \end{aligned}$$

$$\begin{aligned} & \text{scanl } (\oplus) \ e \ [x_0, x_1, x_2] \\ & \quad \downarrow \\ & [e, e \oplus x_0, (e \oplus x_0) \oplus x_1, ((e \oplus x_0) \oplus x_1) \oplus x_2] \end{aligned}$$


And just for comparison, here are the other definitions of *scanl*

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

$$\begin{aligned} \text{scanl} & \quad \text{:: } (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta] \\ \text{scanl } f \ e & = \text{map } (\text{foldl } f \ e) \ . \text{inits} \end{aligned}$$

$$\begin{aligned} \text{inits} & \quad \text{:: } [\alpha] \rightarrow [[\alpha]] \\ \text{inits } [] & \quad = [[]] \\ \text{inits } (x:xs) & = [] : \text{map } (x:) \ (\text{inits } xs) \end{aligned}$$

4.5.3 Scan right

The dual computation is given by *scanr*.

$$\begin{aligned} \text{scanr} &:: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta] \\ \text{scanr } f e &= \text{map } (\text{foldr } f e) . \text{tails} \end{aligned}$$

$$\begin{aligned} \text{scanl} &:: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta] \\ \text{scanl } f e &= \text{map } (\text{foldl } f e) . \text{inits} \end{aligned}$$

The function *tails* returns the **tail segments** of a list. For example,

$$\text{tails } [x_0, x_1, x_2] = [[x_0, x_1, x_2], [x_1, x_2], [x_2], []]$$

$$\text{inits } [x_0, x_1, x_2] = [[], [x_0], [x_0, x_1], [x_0, x_1, x_2]]$$

Note that while *inits* produces a list of **initial segments** in **increasing** order of length, *tails* produces the **tail segments** in **decreasing** order of length. We can define *tails* by

$$\begin{aligned} \text{tails} &:: [\alpha] \rightarrow [[\alpha]] \\ \text{tails } [] &= [[]] \\ \text{tails } (x:xs) &= (x:xs) : \text{tails } xs \end{aligned}$$

$$\begin{aligned} \text{inits} &:: [\alpha] \rightarrow [[\alpha]] \\ \text{inits } [] &= [[]] \\ \text{inits } (x:xs) &= [] : \text{map } (x:) (\text{inits } xs) \end{aligned}$$

The corresponding efficient program for *scanr* is given by

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

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

for reference: a reminder of what *scanr* and *scanl* do

$$\text{scanr } (\oplus) e [x_0, x_1, x_2]$$

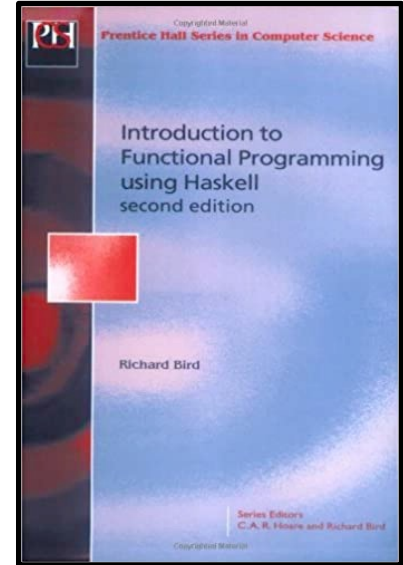


$$[(x_0 \oplus (x_1 \oplus (x_2 \oplus e))), x_1 \oplus (x_2 \oplus e), x_2 \oplus e, e]$$

$$\text{scanl } (\oplus) e [x_0, x_1, x_2]$$



$$[e, e \oplus x_0, (e \oplus x_0) \oplus x_1, ((e \oplus x_0) \oplus x_1) \oplus x_2]$$



Richard Bird



Let's try out *scanl* and *scanr* with (+) and 0



[@philip_schwarz](#)

scanl

applies *foldl* to every **initial segment** of a list

scanr

applies *foldr* to every **tail segment** of a list

inits :: $[\alpha] \rightarrow [[\alpha]]$
inits [] = [[]]
inits (x:xs) = [] : *map* (x:) (*inits* xs)

inits [x₀, x₁, x₂] = [[], [x₀], [x₀, x₁], [x₀, x₁, x₂]]

scanl :: $(\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta]$
scanl f e = *map* (*foldl* f e) . *inits*

scanl (\oplus) e [x₀, x₁, x₂]
↓
[e, e \oplus x₀, (e \oplus x₀) \oplus x₁, ((e \oplus x₀) \oplus x₁) \oplus x₂]

inits [2,3,4] = [[], [2], [2,3], [2,3,4]]

foldl (+) 0 [] = 0
foldl (+) 0 [2] = 2
foldl (+) 0 [2,3] = 5
foldl (+) 0 [2,3,4] = 9

scanl (+) 0 [2,3,4] = [0, 2, 5, 9]

scanl (+) 0 [2,3,4]
↓
[0, 0 + 2, (0 + 2) + 3, ((0 + 2) + 3) + 4]

tails :: $[\alpha] \rightarrow [[\alpha]]$
tails [] = [[]]
tails (x:xs) = (x:xs) : *tails* xs

tails [x₀, x₁, x₂] = [[x₀, x₁, x₂], [x₁, x₂], [x₂], []]

scanr :: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta]$
scanr f e = *map* (*foldr* f e) . *tails*

scanr (\oplus) e [x₀, x₁, x₂]
↓
[(x₀ \oplus (x₁ \oplus (x₂ \oplus e))), x₁ \oplus (x₂ \oplus e), x₂ \oplus e, e]

tails [2,3,4] = [[2,3,4], [3,4], [4], []]

foldr (+) 0 [2,3,4] = 9
foldr (+) 0 [3,4] = 7
foldr (+) 0 [4] = 4
foldr (+) 0 [] = 0

scanr (+) 0 [2,3,4] = [9, 7, 4, 0]

scanr (+) 0 [2,3,4]
↓
[(2 + (3 + (4 + 0))), 3 + (4 + 0), 4 + 0, 0]



Now that we have seen the **iterate** function and the **scanning** functions, let's see how **Sergei Winitzki** describes that fact that **folding**, **iterating** and **scanning** are what we use in **functional programming** to implement **mathematical induction**.

2.5 Summary

We have seen a broad overview of translating **mathematical induction** into **Scala** code.

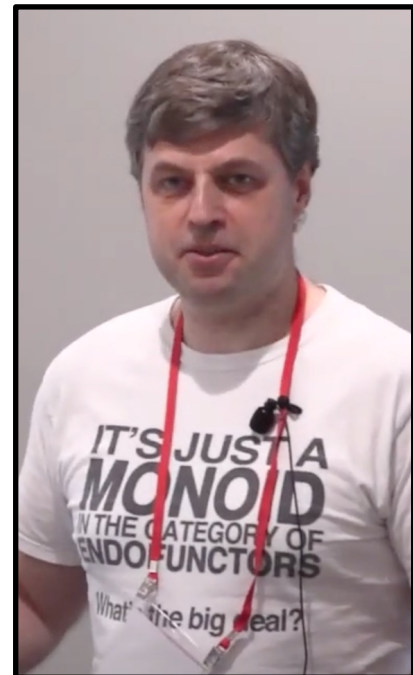
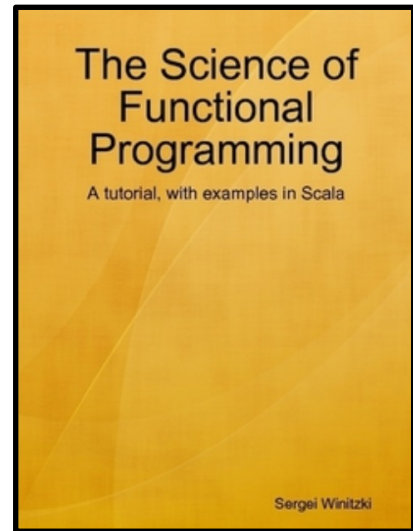
What problems can we solve now?

- Compute **mathematical expressions** involving arbitrary **recursion**.
- Use the **accumulator trick** to enforce **tail recursion**.
- Implement functions with type parameters.
- **Use arbitrary inductive (i.e., recursive) formulas to:**
 - convert **sequences** to single values (**aggregation** or “**folding**”);
 - create new **sequences** from single values (“**unfolding**”);
 - **transform** existing **sequences** into new **sequences**.

Definition by induction	Scala code example
$f([]) = b; f(s++[x]) = g(f(s), x)$	<code>f(xs) = xs.foldLeft(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>

Table 2.1: Implementing **mathematical induction**

Table 2.1 shows **Scala code implementing those tasks**. **Iterative calculations are implemented by translating mathematical induction directly into code**. **In the functional programming paradigm, the programmer does not need to write any loops or use array indices**. **Instead, the programmer reasons about sequences as mathematical values**: “Starting from this value, we get that sequence, then transform it into this other sequence,” etc. This is a **powerful way of working with sequences, dictionaries, and sets**. Many kinds of programming errors (such as an incorrect array index) are avoided from the outset, and the code is **shorter and easier to read** than conventional code written using loops.



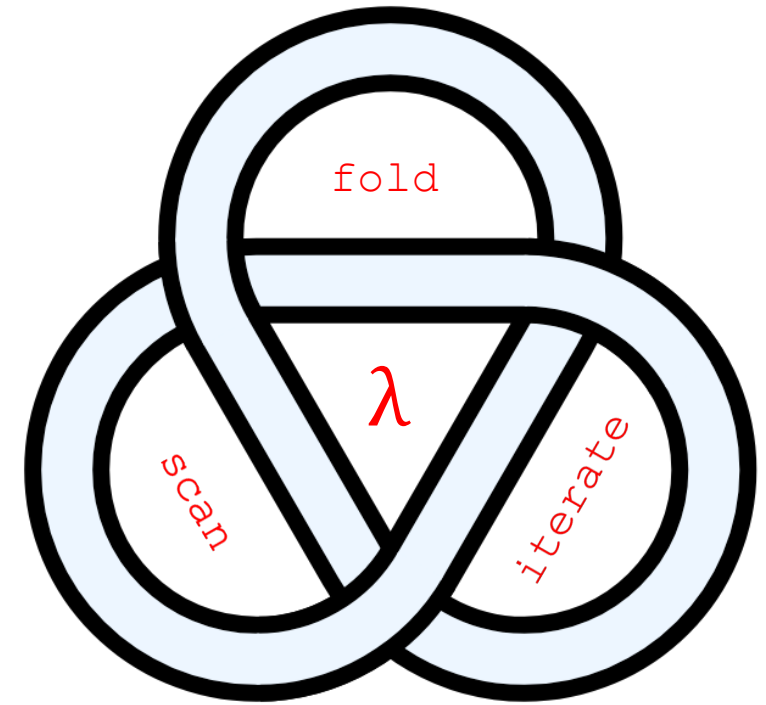
Sergei Winitzki

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



The next slide is a reminder of the **Functional Programming triad** of **folding**, **scanning** and **iteration**, with some very simple examples of their usage in **Scala** and **Haskell**.

Definition by induction	Scala code example
$f([]) = b; f(s++[x]) = g(f(s), x)$	<code>f(xs) = xs.foldLeft(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>



Implementing **mathematical induction**

≡ Haskell

```
Haskell> foldl (+) 0 [1,2,3,4]
10

Haskell> take 4 (iterate (+ 1) 1)
[1,2,3,4]

Haskell> scanl (+) 0 [1,2,3,4]
[0,1,3,6,10]

Haskell>
```

≡ Scala

```
scala> List(1,2,3,4).foldLeft(0)(_+_)
val res0: Int = 10

scala> Stream.iterate(1)(_ + 1).take(4).toList
val res1: List[Int] = List(1, 2, 3, 4)

scala> List(1,2,3,4).scanLeft(0)(_+_)
val res2: List[Int] = List(0, 1, 3, 6, 10)

scala>
```

Remember when we tried to compute the ten-thousandth **Fibonacci** number using the traditional **recursive** definition of the **fibonacci** function, and we got a **stack overflow** error?

```
def fibonacci(n: Int): Long = n match {  
  case 0 => 0L  
  case 1 => 1L  
  case n =>  
    fibonacci(n - 1) + fibonacci(n - 2)  
}
```

```
scala> fibonacci(10_000)  
java.lang.StackOverflowError  
  at fibonacci(<console>:1)  
  at fibonacci(<console>:4)  
  ...
```

With a function like **Factorial**, which makes a single **recursive** call, we can address the problem simply by using the **accumulator technique** to write a **tail recursive** version of the function.

```
def factorial(n: Int): Long = {  
  if (n == 0) 1L  
  else n * factorial(n - 1L)  
}
```

```
faciter 0 acc = acc  
faciter n acc = faciter (n - 1) (n * acc)
```

```
fac n = faciter n 1
```

```
> map fac [0..10]  
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

But sometimes it is not possible to do that. E.g. if a function makes more than one **recursive** call to itself, it is not possible to rewrite those two calls as one.

In the particular case of the **Fibonacci** function, which calls itself **recursively** twice, we can get round the problem by complementing the **accumulator technique** with **tupling**:

```
fibiter 0 (a,b) = a  
fibiter n (a,b) = fibiter (n-1) (b,a+b)  
  
fib n = fibiter n (0,1)
```

```
> map fib [0..10]  
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

On the next slide we conclude this part with [Sergei Winitzki](#) describing the limitations of the **accumulator technique**.



 @philip_schwarz



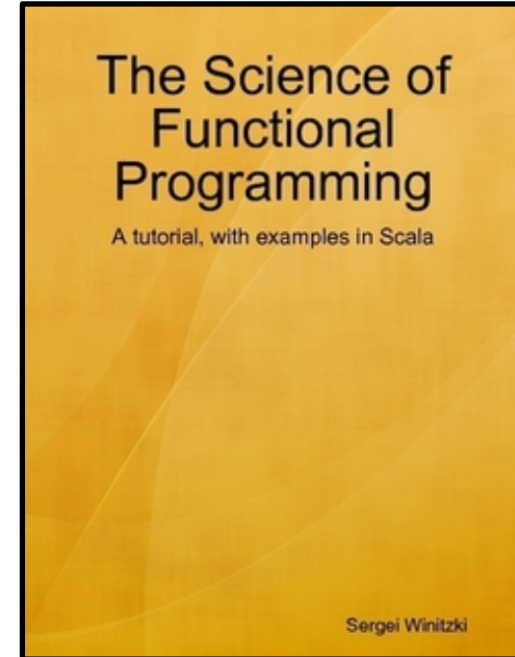
Sergei Winitzki

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

We cannot implement a **non-tail-recursive** function without **stack overflow** (i.e., without unlimited growth of **intermediate expressions**).

The **accumulator trick** does not always work! In some cases, it is impossible to implement **tail recursion** in a given recursive computation. An example of such a computation is the “**merge-sort**” algorithm where the function body must contain two recursive calls within a single expression. (It is impossible to rewrite two recursive calls as one.)

What if our **recursive** code cannot be transformed into **tail-recursive** code via the **accumulator trick**, but the **recursion depth** is so large that **stack overflows** occur? There exist special tricks (e.g., “**continuations**” and “**trampolines**”) that convert **non-tail-recursive** code into **iterative** code without **stack overflows**. Those techniques are beyond the scope of this chapter.





That's all for Part 4. See you in Part 5.