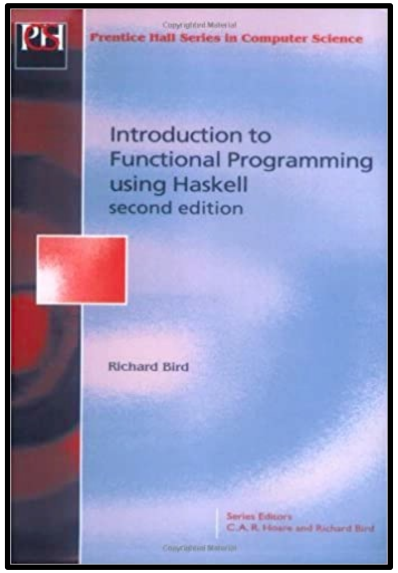
The **Functional Programming Triad** of **Folding**, **Scanning** and **Iteration**

a first example in **Scala** and **Haskell**

Polyglot **FP** for **F**un and **P**rofit



Prentice Hall Series in Computer Science

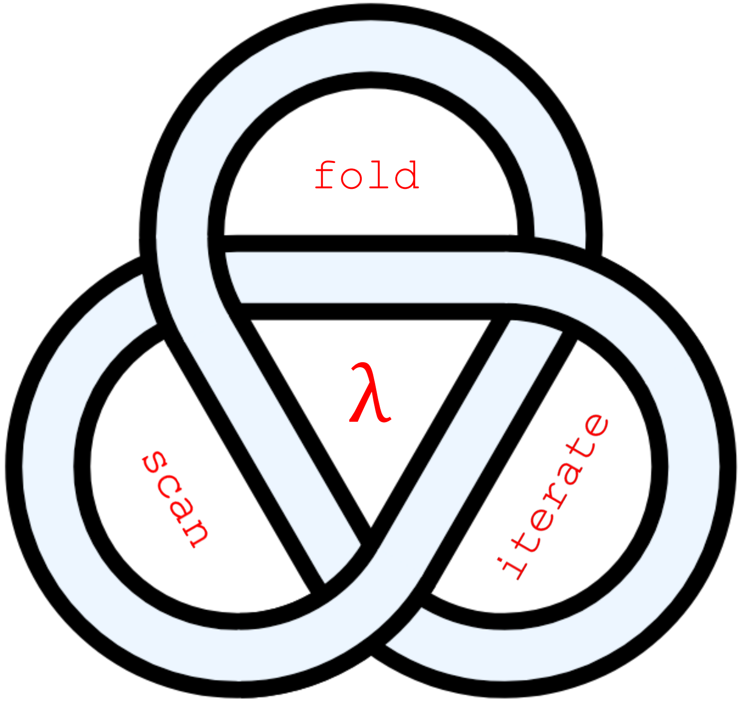
Introduction to
Functional Programming
using Haskell
second edition

Richard Bird

Series Editors
C.A.R. Hoare and Richard Bird

Richard Bird

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

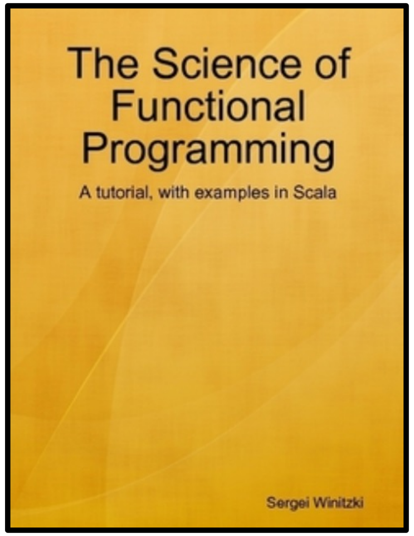


slides by 

slideshare

@philip_schwarz

https://www.slideshare.net/pjschwarz



The Science of
Functional
Programming

A tutorial, with examples in Scala

Sergei Winitzki

Sergei Winitzki

sergei-winitzki-11a6431

@philip_schwarz

This slide deck can work both as an **aide mémoire** (**memory jogger**), or as a first (not completely trivial) example of using **left folds**, **left scans** and **iteration** to implement **mathematical induction**.

We first look at the implementation, using a **left fold**, of a `digits-to-int` function that **converts a sequence of digits into a whole number**.

Then we look at the implementation, using the **iterate** function, of an `int-to-digits` function **that converts an integer into a sequence of digits**. In **Scala** this function is very readable because the signatures of functions provided by **collections** permit the **piping** of such functions with zero **syntactic overhead**. In **Haskell**, there is some **syntactic sugar** that can be used to achieve the same readability, so we look at how that works.

We then set ourselves a simple task involving `digits-to-int` and `int-to-digits`, and write a function whose logic can be simplified with the introduction of a **left scan**.

Let's begin, on the next slide, by looking at how **Richard Bird** describes the `digits-to-int` function, which he calls *decimal*.

Suppose we want a function *decimal* that takes a list of digits and returns the corresponding decimal number; thus

$$decimal\ [x_0, x_1, \dots, x_\mathrm{n}] = \sum_{k=0}^{n} x_k 10^{(n-k)}$$

It is assumed that the most significant digit comes first in the list. One way to compute *decimal* efficiently is by a process of multiplying each digit by ten and adding in the following digit. For example

$$decimal\ [x_0, x_1, x_2] = 10 \times (10 \times (10 \times 0 + x_0) + x_1) + x_2$$

This decomposition of a sum of powers is known as ***Horner's* rule**.

Suppose we define $\oplus$ by $n \oplus x = 10 \times n + x$. Then we can rephrase the above equation as

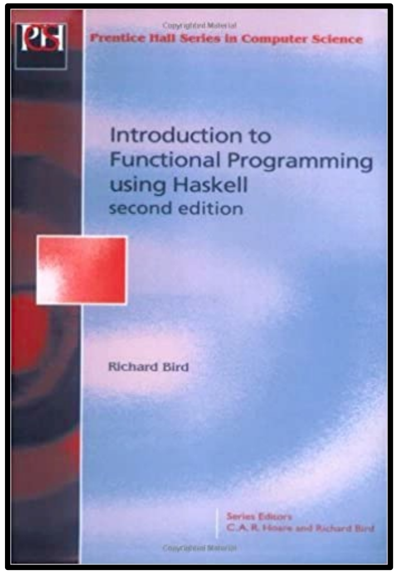
$$decimal\ [x_0, x_1, x_2] = ((0 \oplus x_0) \oplus x_1) \oplus x_2$$

This example motivates the introduction of a second fold operator called *foldl* (pronounced 'fold left'). Informally:

$$foldl\ (\oplus)\ e\ [x_0, x_1, \dots, xn_{\_\,1}] = (\dots((e \oplus x_0) \oplus x_1) \dots) \oplus x_{n\,\_\,1}$$

The parentheses group from the left, which is the reason for the name. The full definition of *foldl* is

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



Prentice Hall Series in Computer Science

Introduction to Functional Programming using Haskell
second edition

Richard Bird

Series Editors
C.A.R. Hoare and Richard Bird

Richard Bird

On the next slide we look at how **Sergei Winitzki** describes the `digits-to-int` function, and how he implements it in **Scala**.

Example 2.2.5.3 Implement the function `digits-to-int` using `foldLeft`.

…

The required computation can be written as the formula

$$r = \sum_{k=0}^{n-1} d_k * 10^{n-1-k}.$$

…

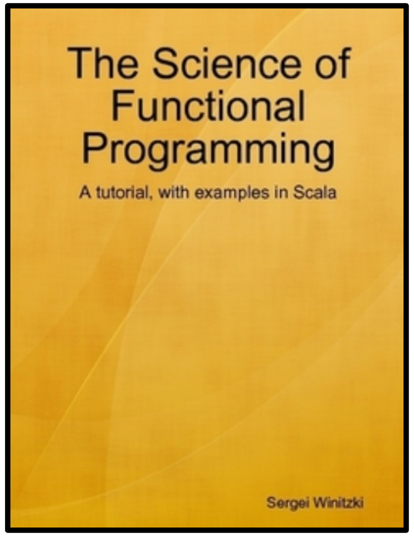
Solution The inductive definition of `digitsToInt`

- For an empty sequence of digits, `Seq()`, the result is 0. This is a convenient base case, even if we never call `digitsToInt` on an empty sequence.
- If `digitsToInt(xs)` is already known for a sequence `xs` of digits, and we have a sequence `xs :+ x` with one more digit `x`, then

```
digitsToInt(xs :+ x) = digitsToInt(xs) * 10 + x
```

is directly translated into code:

```
def digitsToInt(d: Seq[Int]): Int = d.foldLeft(0){ (n, x) => n * 10 + x }
```


The Science of Functional Programming
A tutorial, with examples in Scala
Sergei Winitzki

Sergei Winitzki

in sergei-winitzki-11a6431

Here is a quick refresher on **fold left**

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

$$\begin{aligned} & foldl\ (\oplus)\ \ e\ \ [x_0, x_1, x_2] \\ = \ & foldl\ (\oplus)\ (e \oplus x_0)\ \ [x_1, x_2] \\ = \ & foldl\ (\oplus)\ \big((e \oplus x_0) \oplus x_1\big)\ \ [x_2] \\ = \ & foldl\ (\oplus)\ \big(((e \oplus x_0) \oplus x_1) \oplus x_2\big)\ \ [\,] \\ = \ & ((e \oplus x_0) \oplus x_1) \oplus x_2 \end{aligned}$$

$$\begin{aligned} & foldl\ (\oplus)\ \ 0\ \ [1,2,3] \\ = \ & foldl\ (\oplus)\ (0 \oplus 1)\ \ [2,3] \\ = \ & foldl\ (\oplus)\ \big((0 \oplus 1) \oplus 2\big)\ \ [3] \\ = \ & foldl\ (\oplus)\ \big(((0 \oplus 1) \oplus 2) \oplus 3\big)\ \ [\,] \\ = \ & ((0 \oplus 1) \oplus 2) \oplus 3 \end{aligned}$$

$$foldl\ (\oplus)\ e\ [x_0, x_1, \dots, xn] = (\dots((e \oplus x_0) \oplus x_1) \dots) \oplus x_n$$

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

So let's implement the **digits-to-int** function in **Haskell**.

```
(⊕) :: Int -> Int -> Int
(⊕) n d = 10 * n + d

digits_to_int :: [Int] -> Int
digits_to_int = foldl (⊕) 0
```

And now in **Scala**.

```
extension (n:Int)
  def ⊕ (d:Int) = 10 * n + d

def digitsToInt(ds: Seq[Int]): Int =
  ds.foldLeft(0)(_⊕_)
```

```
assert( digitsToInt(List(5,3,7,4)) == 5374)
```

$$decimal\ [x_0, x_1, \dots, x_n] = \sum_{k=0}^{n} x_k 10^{(n-k)}$$

```
assert( (150 ⊕ 7) == 1507 )
```

Now we turn to **int-to-digits**, a function that is the opposite of **digits-to-int**, and one that can be implemented using the **iterate** function.

In the next two slides, we look at how **Sergei Winitzki** describes **digits-to-int** (which he calls `digitsOf`) and how he implements it in **Scala**.

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.

…

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 (n_0 = 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:

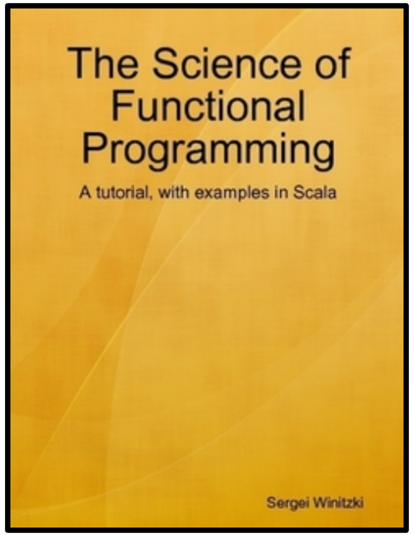
- **Base case**: $n_0 = n$, where n is the given initial integer.
- **Inductive step**: $n_{k+1} = \left\lfloor \frac{n_k}{10} \right\rfloor$ for k = 1, 2, ...

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

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`.



The Science of Functional Programming

A tutorial, with examples in Scala

Sergei Winitzki

Sergei Winitzki

sergei-winitzki-11a6431

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:

…

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

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

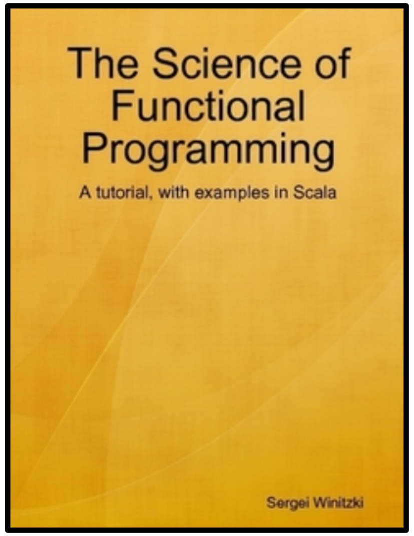
and shows a close correspondence to a definition by mathematical induction. The base case is the first value, `init`, and the inductive step is a function, `next`, that computes the next element from the previous one. **It is a general way of creating sequences whose length is not determined in advance.**

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
}
```



The Science of Functional Programming
A tutorial, with examples in Scala
Sergei Winitzki

Sergei Winitzki

in sergei-winitzki-11a6431

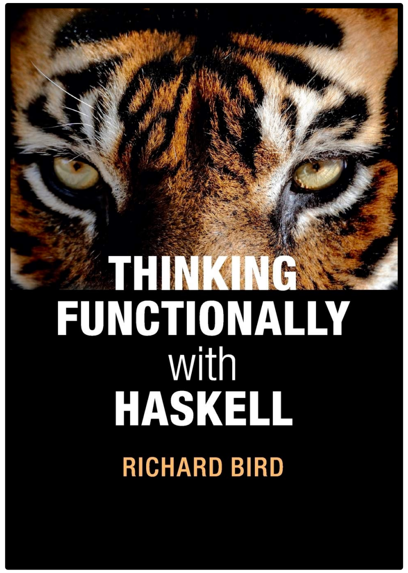
Here is how **Richard Bird** describes the **iterate** function

Richard Bird

the prelude function **iterate** returns an **infinite list**:

$$\mathit{iterate} :: (a \rightarrow a) \rightarrow a \rightarrow [a]$$
$$\mathit{iterate}\ f\ x = x : \mathit{iterate}\ f\ (f\ x)$$

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



Here on the left I had a go at a **Haskell** implementation of the **int-to-digits** function (we are not handling cases like n=0 or negative n), and on the right, the same logic in **Scala**.

```haskell
int_to_digits :: Int -> [Int]
int_to_digits n =
  reverse (
    map (\x -> mod x 10) (
      takeWhile (\x -> x /= 0) (
        iterate (\x -> div x 10)
          n
      )
    )
  )
```

I find the **Scala** version slightly easier to understand, because the functions that we are calling appear in the order in which they are executed. Contrast that with the **Haskell** version, in which the function invocations occur in the opposite order.

```scala
import LazyList.iterate

def intToDigits(n: Int): Seq[Int] =
  iterate(n) (_ / 10 )
    .takeWhile ( _ != 0 )
    .map ( _ % 10 )
    .toList
    .reverse
```

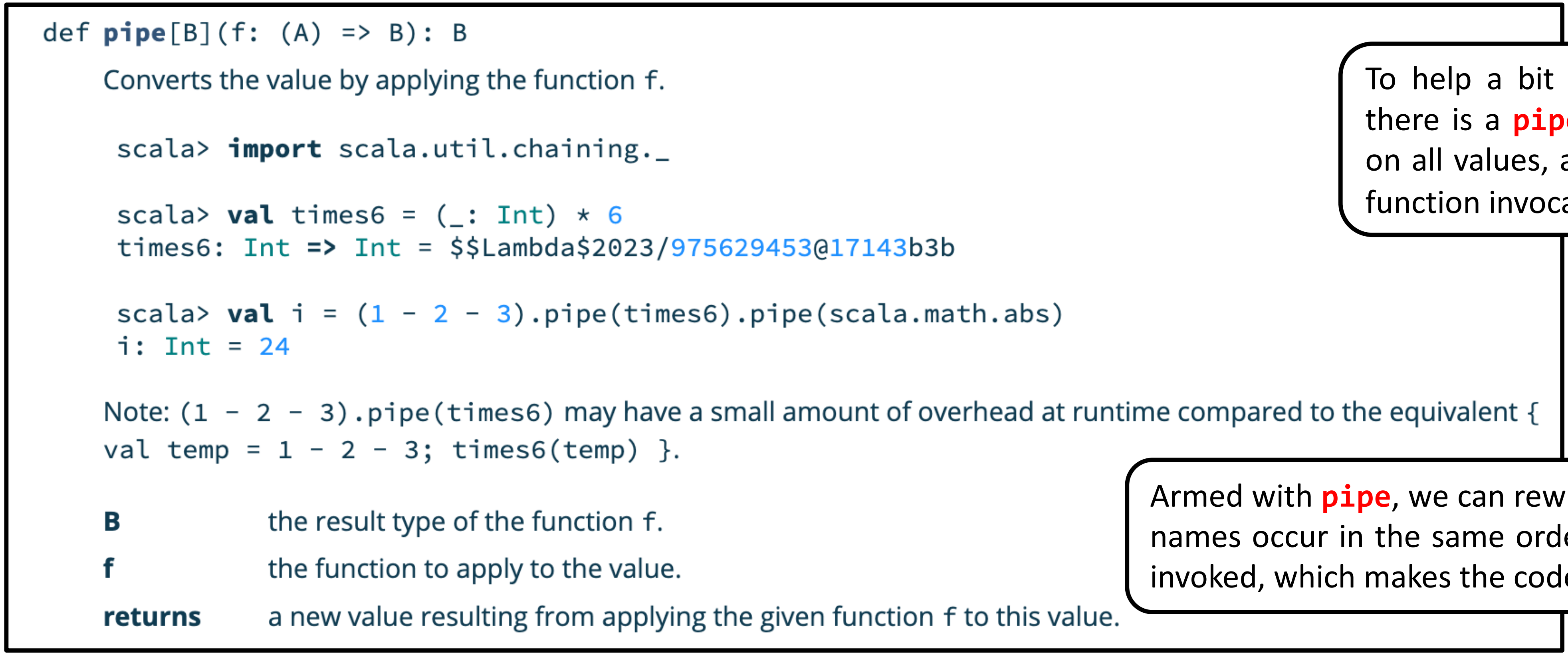
While the ‘fluent’ **chaining** of function calls on **Scala** collections is very convenient, when using other functions, we face the same problem that we saw in **Haskell** on the previous slide. e.g. in the following code, **square** appears first, even though it is executed last, and vice versa for **inc**.

@philip_schwarz

```
square(twice(inc(3)))
```

```
def inc(n: Int): Int = n + 1
def twice(n: Int): Int = n * 2
def square(n: Int): Int = n * n
```

```
assert(square(twice(inc(3))) == 64)
```



`def pipe[B](f: (A) => B): B`

Converts the value by applying the function f.

```
scala> import scala.util.chaining._

scala> val times6 = (_: Int) * 6
times6: Int => Int = $$Lambda$2023/975629453@17143b3b

scala> val i = (1 - 2 - 3).pipe(times6).pipe(scala.math.abs)
i: Int = 24
```

Note: `(1 - 2 - 3).pipe(times6)` may have a small amount of overhead at runtime compared to the equivalent `{ val temp = 1 - 2 - 3; times6(temp) }`.

B	the result type of the function f.
f	the function to apply to the value.
returns	a new value resulting from applying the given function f to this value.

To help a bit with that problem, in **Scala** there is a **pipe** function which is available on all values, and which allows us to order function invocations in the ‘right’ order.

Armed with **pipe**, we can rewrite the code so that function names occur in the same order in which the functions are invoked, which makes the code more understandable.

```
assert ((3 pipe inc pipe twice pipe square) == 64)
```

```
3 pipe inc
  pipe twice
  pipe square
```

What about in **Haskell**? First of all, in **Haskell** there is a **function application** operator called **$**, which we can sometimes use to omit parentheses

```
($) :: forall r a (b :: TYPE r) . (a -> b) -> a -> b
```

base Prelude Data.Function GHC.Base

Application operator. This operator is redundant, since ordinary application `(f x)` means the same as `(f $ x)`.

However, **$** has low, right-associative binding precedence, so it sometimes allows parentheses to be omitted; for example:

```
f $ g $ h x = f (g (h x))
```

For beginners, the **$** often makes **Haskell** code more difficult to parse. In practice, the **$** operator is used frequently, and you'll likely find you prefer using it over many parentheses. There's nothing magical about $; if you look at its type signature, you can see how it works:

```
($) :: (a -> b) -> a -> b
```

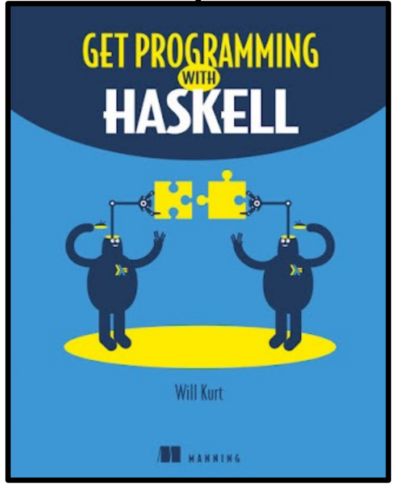
The arguments are just a function and a value. The trick is that **$** is a binary operator, so it has lower precedence than the other functions you're using. Therefore, the argument for the function will be evaluated as though it were in parentheses.

```
int_to_digits :: Int -> [Int]
int_to_digits n =
  reverse (
    map (\x -> mod x 10) (
      takeWhile (\x -> x /= 0) (
        iterate (\x -> div x 10)
          n
      )
    )
  )
```

Armed with **$**, we can simplify our function as follows

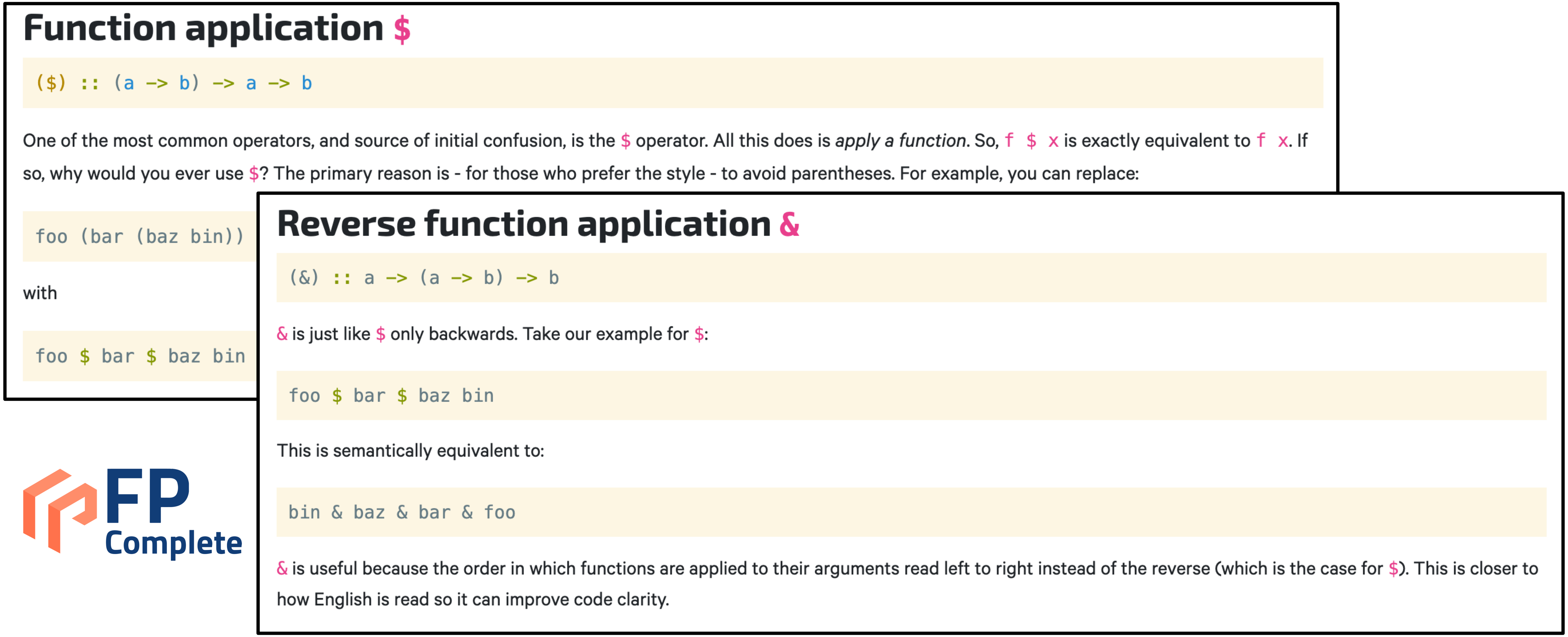
simplify

```
int_to_digits :: Int -> [Int]
int_to_digits n =
  reverse $ map (\x -> mod x 10)
          $ takeWhile (\x -> x > 0)
          $ iterate (\x -> div x 10)
          $ n
```



But there is more we can do. In addition to the **function application** operator **$**, in **Haskell** there is also a **reverse function application** operator **&**.

https://www.fpcomplete.com/haskell/tutorial/operators/



Function application $

```
($) :: (a -> b) -> a -> b
```

One of the most common operators, and source of initial confusion, is the `$` operator. All this does is *apply a function*. So, `f $ x` is exactly equivalent to `f x`. If so, why would you ever use `$`? The primary reason is - for those who prefer the style - to avoid parentheses. For example, you can replace:

```
foo (bar (baz bin))
```

with

```
foo $ bar $ baz bin
```

Reverse function application &

```
(&) :: a -> (a -> b) -> b
```

`&` is just like `$` only backwards. Take our example for `$`:

```
foo $ bar $ baz bin
```

This is semantically equivalent to:

```
bin & baz & bar & foo
```

`&` is useful because the order in which functions are applied to their arguments read left to right instead of the reverse (which is the case for `$`). This is closer to how English is read so it can improve code clarity.

Thanks to the & operator, we can rearrange our `int_to_digits` function so that it is as readable as the **Scala** version.

```
int_to_digits :: Int -> [Int]
int_to_digits n =
  reverse $ map (\x -> mod x 10)
          $ takeWhile (\x -> x > 0)
          $ iterate (\x -> div x 10)
          $ n
```

```
int_to_digits :: Int -> [Int]
int_to_digits n =
  n & iterate (\x -> div x 10)
    & takeWhile (\x -> x > 0)
    & map (\x -> mod x 10)
    & reverse
```

```
def intToDigits(n: Int): Seq[Int] =
  iterate(n) (_ / 10 )
    .takeWhile ( _ != 0 )
    .map ( _ % 10 )
    .toList
    .reverse
```

There is one school of thought according to which the choice of names for $ and & make **Haskell** hard to read for newcomers, that it is better if $ and & are instead named <| and |>.

https://hackage.haskell.org/package/flow-1.0.21/docs/Flow.html

flow-1.0.21: Write more understandable Haskell.

Flow

Here is an example of how |> and <| improve readability

Flow provides **operators for writing more understandable Haskell**. It is **an alternative to some common idioms like** (**$**) **for function application and** (.) **for function composition**.

…

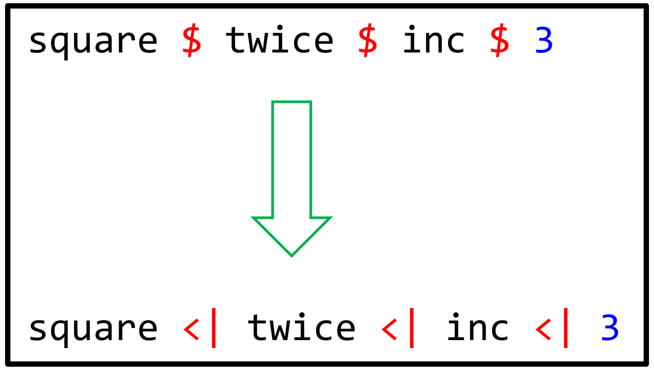
Rationale

I think that **Haskell can be hard to read**. It has two operators for applying functions. Both are not really necessary and only serve to reduce parentheses. But they make code hard to read. **People who do not already know Haskell have no chance of guessing what foo $ bar** or **baz & qux mean**.

…

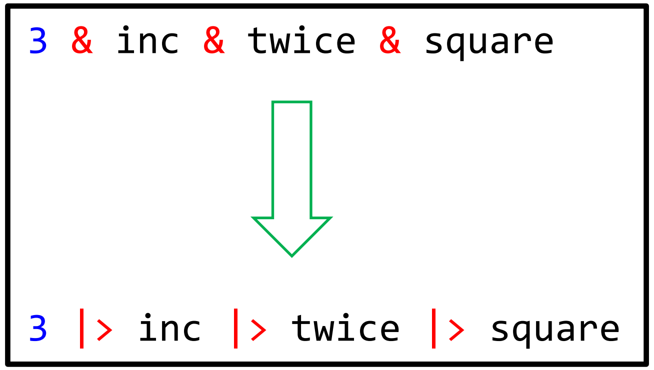
I think we can do better. **By using directional operators, we can allow readers to move their eye in only one direction, be that left-to-right or right-to-left**. And by using idioms common in other programming languages, we can allow people who aren't familiar with **Haskell** to guess at the meaning.

So **instead of** (**$**), **I propose** (**<|**). **It is a pipe, which anyone who has touched a Unix system should be familiar with. And it points in the direction it sends arguments along. Similarly, replace** (**&**) **with** (**|>**). …



```
square $ twice $ inc $ 3
```

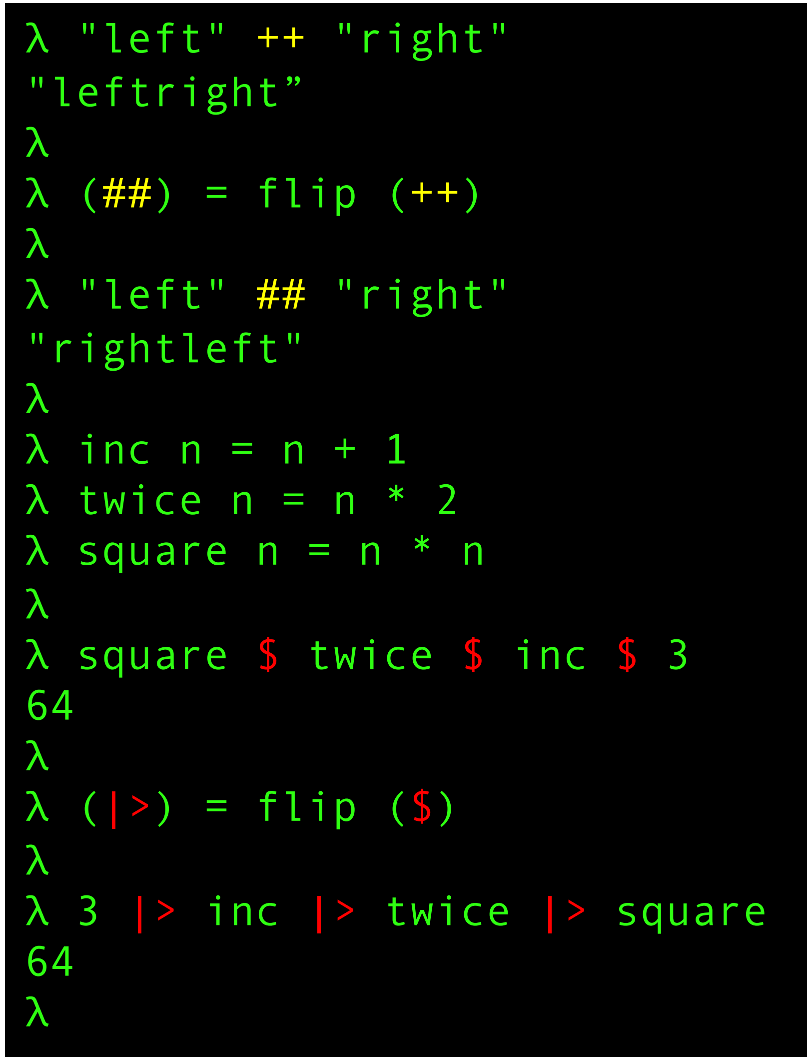
```
square <| twice <| inc <| 3
```



```
3 & inc & twice & square
```

```
3 |> inc |> twice |> square
```

Since & is just the reverse of $, we can define |> ourselves simply by flipping $



```
λ "left" ++ "right"
"leftright"
λ
λ (##) = flip (++)
λ
λ "left" ## "right"
"rightleft"
λ
λ inc n = n + 1
λ twice n = n * 2
λ square n = n * n
λ
λ square $ twice $ inc $ 3
64
λ
λ (|>) = flip ($)
λ
λ 3 |> inc |> twice |> square
64
λ
```

And here is how our function looks using |>.

@philip_schwarz

```
int_to_digits :: Int -> [Int]
int_to_digits n =
  n |> iterate (\x -> div x 10)
    |> takeWhile (\x -> x > 0)
    |> map (\x -> mod x 10)
    |> reverse
```

Now let's set ourselves the following task. Given a positive integer N with n digits, e.g. the five-digit number 12345, we want to compute the following:

```
[(0,0),(1,1),(12,3),(123,6),(1234,10),(12345,15)]
```

i.e. we want to compute a list of pairs $p_0, p_1, \dots, p_n$ with p_k being (N_k , $N_{k\Sigma}$), where N_k is the integer number formed by the first k digits of N, and $N_{k\Sigma}$ is the sum of those digits. We can use our **int_to_digits** function to convert N into its digits $d_1, d_2, \dots, d_n$:

```
λ int_to_digits 12345
[1,2,3,4,5]
λ
```

And we can use **digits_to_int** to turn digits $d_1, d_2, \dots, d_k$ into N_k, e.g. for $k = 3$:

```
λ digits_to_int [1,2,3]
123
λ
```

How can we generate the following sequences of digits ?

$$[\,[\,]\,, [d_1]\,, [d_1, d_2]\,, [d_1, d_2, d_3]\,, \dots, [d_1, d_2, d_3, \dots, d_n]\,]$$

As we'll see on the next slide, that is exactly what the **inits** function produces when passed $[d_1, d_2, d_3, \dots, d_n]$!

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

Here is a definition for **inits**.

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

And here is what **inits** produces, i.e. the list of all **initial segments** of a list.

So we can apply **`inits`** to $[d_1, d_2, d_3, \dots, d_n]$ to generate the following:

$$[\,[\,], [d_1], [d_1, d_2], [d_1, d_2, d_3], \dots, [d_1, d_2, d_3, \dots, d_n]\,]$$

e.g.

```
λ inits [1,2,3,4]
[[],[1],[1,2],[1,2,3],[1,2,3,4]]
λ
```

So if we map **digits_to_int** over the initial segments of $[d_1, d_2, d_3, \ldots, d_n]$, i.e

$$[\,[\,], [d_1], [d_1, d_2], [d_1, d_2, d_3], \ldots, [d_1, d_2, d_3, \ldots, d_n]\,]$$

we obtain a list containing $N_0, N_1, \ldots, N_n$, e.g.

```
λ map digits_to_int (inits [1,2,3,4,5])
[0,1,12,123,1234,12345]
λ
```

```
(⊕) :: Int -> Int -> Int
(⊕) n d = 10 * n + d

digits_to_int :: [Int] -> Int
digits_to_int = foldl (⊕) 0
```

What we need now is a function **digits_to_sum** which is similar to **digits_to_int** but which instead of converting a list of digits $[d_1, d_2, d_3, \ldots, d_k]$ into N_k, i.e. the number formed by those digits, it turns the list into $N_{k\Sigma}$, i.e. the sum of the digits. Like **digits_to_int**, the **digits_to_sum** function can be defined using a **left fold**:

```
digits_to_sum :: [Int] -> Int
digits_to_sum = foldl (+) 0
```

Let's try it out:

```
λ digits_to_sum [1,2,3,4]
10
λ
```

Now if we map **digits_to_sum** over the initial segments of $[d_1, d_2, d_3, \ldots, d_n]$, we obtain a list containing $N_{0\Sigma}, N_{1\Sigma}, \ldots, N_{n\Sigma}$, e.g.

```
λ map digits_to_sum (inits [1,2,3,4,5])
[0,1,3,6,10,15]
λ
```

So here is how our complete program looks at the moment.

@philip_schwarz

```
int_to_digits :: Int -> [Int]
int_to_digits n =
  n |> iterate (\x -> div x 10)
    |> takeWhile (\x -> x > 0)
    |> map (\x -> mod x 10)
    |> reverse
```

```
(⊕) :: Int -> Int -> Int
(⊕) n d = 10 * n + d

digits_to_int :: [Int] -> Int
digits_to_int = foldl (⊕) 0
```

```
digits_to_sum :: [Int] -> Int
digits_to_sum = foldl (+) 0
```

```
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = map digits_to_int segments
                  sums = map digits_to_sum segments
                  segments = inits (int_to_digits n)
```

```
λ convert 12345
[(0,0),(1,1),(12,3),(123,6),(1234,10),(12345,15)]
λ
```

What we are going to do next is see how, by using the **scan left** function, we are able to simplify the definition of **convert** which we just saw on the previous slide.

As a quick refresher of (or introduction to) the **scan left** function, in the next two slides we look at how **Richard Bird** describes the function.

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,

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

In particular, $scanl\ (+)\ 0$ computes the list of accumulated sums of a list of numbers, and $scanl\ (\times)\ 1\ [1..n]$ 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,

$$inits\ [x_0, x_1, x_2] = [[\,], [x_0], [x_0, x_1], [x_0, x_1, x_2]]$$

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

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

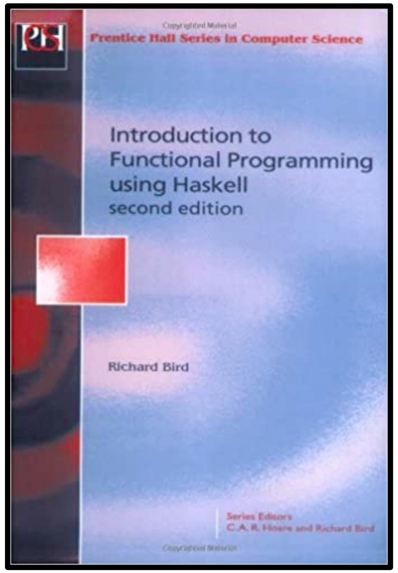
The function $inits$ can be defined more succinctly as an instance of $foldr$:

$$inits = foldr\ f\ [[\,]]\ where\ f\ x\ xss = [\,] : map\ (x\!:)\ xss$$

Now we define

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

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

$foldl\ f\ e$ on a list of length k and, since the **initial segments** of a list of length n are lists with lengths $0,1,\ldots,n$, the function f is applied about $n^2/2$ times in total.

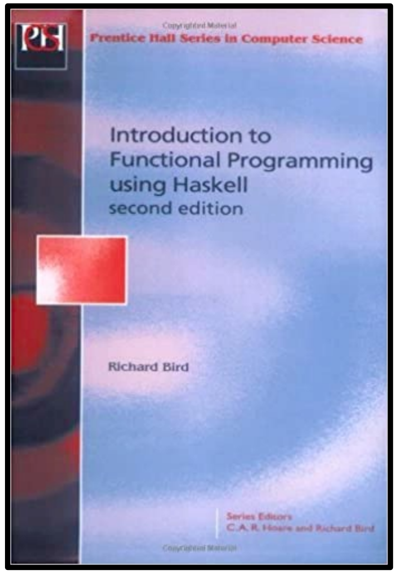
Let us now synthesise a more efficient program. The synthesis is by an **induction argument** on xs so we lay out the calculation in the same way.

<…not shown…>

In summary, we have derived

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

This program is **more efficient** in that function f is applied exactly n times on a list of length n.



Prentice Hall Series in Computer Science

Introduction to
Functional Programming
using Haskell
second edition

Richard Bird

Series Editors
C.A.R. Hoare and Richard Bird

Note the similarities and differences between $scanl$ and $foldl$, e.g. the left hand sides of their equations are the same, and their signatures are very similar, but $scanl$ returns $[\beta]$ rather than β and while $foldl$ is **tail recursive**, $scanl$ isn't.

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

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

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

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

Richard Bird

On the next slide, a very simple example of using **scanl**, and a reminder of how the result of **scanl** relates to the result of **inits** and **foldl**.

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

$$
\begin{aligned}
& foldl\ (\oplus)\ e\ [x_0, x_1, x_2] \\
= \;& foldl\ (\oplus)\ (e \oplus x_0)\ [x_1, x_2] \\
= \;& foldl\ (\oplus)\ \big((e \oplus x_0) \oplus x_1\big)\ [x_2] \\
= \;& foldl\ (\oplus)\ (((e \oplus x_0) \oplus x_1) \oplus x_2)\ [\,] \\
= \;& ((e \oplus x_0) \oplus x_1) \oplus x_2
\end{aligned}
$$

$$
\begin{aligned}
& foldl\ (\oplus)\ e\ [x_0, x_1, \ldots, xn_{\_1}] = \\
& \quad (\ldots((e \oplus x_0) \oplus x_1) \ldots) \oplus x_{n\_1}
\end{aligned}
$$

$$
\begin{aligned}
& foldl\ (+)\ 0\ [\,] = 0 \\
& foldl\ (+)\ 0\ [2] = 2 \\
& foldl\ (+)\ 0\ [2,3] = 5 \\
& foldl\ (+)\ 0\ [2,3,4] = 9
\end{aligned}
$$

$$inits\ [x_0, x_1, x_2] = [[\,], [x_0], [x_0, x_1], [x_0, x_1, x_2]]$$

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

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

$$
\begin{gathered}
scanl\ (\oplus)\ e\ [x_0, x_1, x_2] \\
\Downarrow \\
[e, e \oplus x_0, (e \oplus x_0) \oplus x_1, ((e \oplus x_0) \oplus x_1) \oplus x_2]
\end{gathered}
$$

$$
\begin{gathered}
scanl\ (+)\ 0\ [2,3,4] \\
\Downarrow \\
[0, 0 + 2, (0 + 2) + 3, \big((0 + 2) + 3\big) + 4]
\end{gathered}
$$

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

After that refresher of (introduction to) the **scanl** function, let’s see how it can help us simplify our definition of the **convert** function.

The first thing to do is to take the definition of **convert** and inline its invocations of `digits_to_int` and `digits_to_sum`:

```haskell
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = map digits_to_int segments
                  sums = map digits_to_sum segments
                  segments = inits (int_to_digits n)
```

refactor

```haskell
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = map foldl (⊕) 0 segments
                  sums = map foldl (+) 0 segments
                  segments = inits (int_to_digits n)
```

Now let’s extract `(int_to_digits n)` into `digits` and inline `segments`.

```haskell
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = map foldl (⊕) 0 segments
                  sums = map foldl (+) 0 segments
                  segments = inits (int_to_digits n)
```

refactor

```haskell
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = map foldl (⊕) 0 (inits digits)
                  sums = map foldl (+)  0 (inits digits)
                  digits = int_to_digits n
```

As we saw earlier, mapping **foldl** over the result of applying **inits** to a list, is just applying **scanl** to that list, so let’s simplify **convert** by calling **scanl** rather than mapping **foldl**.

$$scanl\ f\ e\ =map\ (foldl\ f\ e)\ .\ inits$$

```haskell
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = map foldl (⊕) 0 (inits digits)
                  sums = map foldl (+)  0 (inits digits)
                  digits = int_to_digits n
```

refactor

```haskell
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = scanl (⊕) 0 digits
                  sums = scanl (+)  0 digits
                  digits = int_to_digits n
```

@philip_schwarz

The suboptimal thing about our current definition of **convert** is that it does two **left scans** over the same list of digits.

```
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = scanl (⊕) 0 digits
                  sums = scanl (+)  0 digits
                  digits = int_to_digits n
```

Can we refactor it to do a single **scan**? Yes, by using **tupling**, i.e. by changing the function that we pass to **scanl** from a function like **(⊕)** or **(+)**, which computes a single result, to a function, let's call it **next**, which uses those two functions to compute two results

```
convert :: Int -> [(Int,Int)]
convert n = scanl next (0, 0) digits
            where next (number, sum) digit = (number ⊕ digit, sum + digit)
                  digits = int_to_digits n
```

On the next slide, we inline **digits** and compare the resulting **convert** function with our initial version, which invoked **scanl** twice.

Here is our first definition of **convert**

```
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = map digits_to_int segments
                  sums = map digits_to_sum segments
                  segments = inits (int_to_digits n)
```

And here is our refactored version, which uses a **left scan**.

```
convert :: Int -> [(Int,Int)]
convert n = scanl next (0, 0) (int_to_digits n)
            where next (number, sum) digit = (number ⊕ digit, sum + digit)
```

The next slide shows the complete **Haskell** program, and next to it, the equivalent **Scala** program.

```
int_to_digits :: Int -> [Int]
int_to_digits n =
  n |> iterate (\x -> div x 10)
    |> takeWhile (\x -> x > 0)
    |> map (\x -> mod x 10)
    |> reverse
```

```
(⊕) :: Int -> Int -> Int
(⊕) n d = 10 * n + d
```

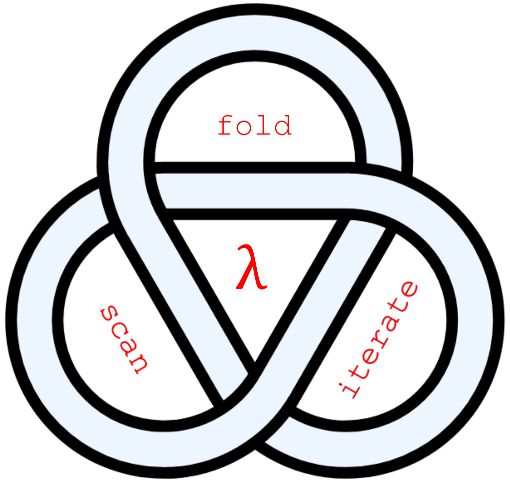
```
digits_to_int :: [Int] -> Int
digits_to_int = foldl (⊕) 0
```

```
digits_to_sum :: [Int] -> Int
digits_to_sum = foldl (+) 0
```

```
convert :: Int -> [(Int,Int)]
convert n = zip nums sums
            where nums = map digits_to_int segments
                  sums = map digits_to_sum segments
                  segments = inits (int_to_digits n)
```

```
convert' :: Int -> [(Int,Int)]
convert' n = scanl next (0, 0) (int_to_digits n)
            where next (number, sum) digit =
              (number ⊕ digit, sum + digit)
```

```
λ convert 1234
[(0,0),(1,1),(12,3),(123,6),(1234,10)]
λ
```



```
def intToDigits(n: Int): Seq[Int] =
  iterate(n) (_ / 10 )
    .takeWhile ( _ != 0 )
    .map ( _ % 10 )
    .toList
    .reverse
```

```
extension (n:Int)
  def ⊕ (d:Int) = 10 * n + d
```

```
def digitsToInt(ds: Seq[Int]): Int =
  ds.foldLeft(0)(_⊕_)
```

```
def digitsToSum(ds: Seq[Int]): Int =
  ds.foldLeft(0)(_+_)
```

```
def `convert⌛`(n: Int): Seq[(Int,Int)] =
  val segments = intToDigits(n).inits.toList.reverse
  val nums = segments map digitsToInt
  val sums = segments map digitsToSum
  nums zip sums
```

```
def `convert⚡`(n: Int): Seq[(Int,Int)] =
  val next: ((Int,Int),Int) => (Int,Int) =
    case ((number, sum), digit) =>
      (number ⊕ digit, sum + digit)
  intToDigits(n).scanLeft((0,0))(next)
```

```
assert(intToDigits(1234) == List(1,2,3,4));  assert((123 ⊕ 4) == 1234)
assert(digitsToInt(List(1,2,3,4)) == 1234)
assert(digitsToSum(List(1,2,3,4)) == 10)
assert(`convert⌛`(1234) == List((0,0),(1,1),(12,3),(123,6),(1234,10)))
assert(`convert⚡`(1234) == List((0,0),(1,1),(12,3),(123,6),(1234,10)))
```

In the next slide we conclude this deck with **Sergei Winitzki**‘s recap of how in **functional programming** we implement **mathematical induction** using **folding**, **scanning** and **iteration**.

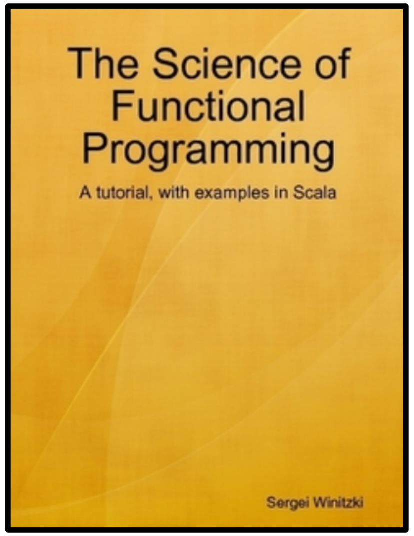
Use arbitrary inductive (i.e., recursive) formulas to:

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

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.



The Science of Functional Programming

A tutorial, with examples in Scala

Sergei Winitzki

Sergei Winitzki

in sergei-winitzki-11a6431