Folding Unfolded

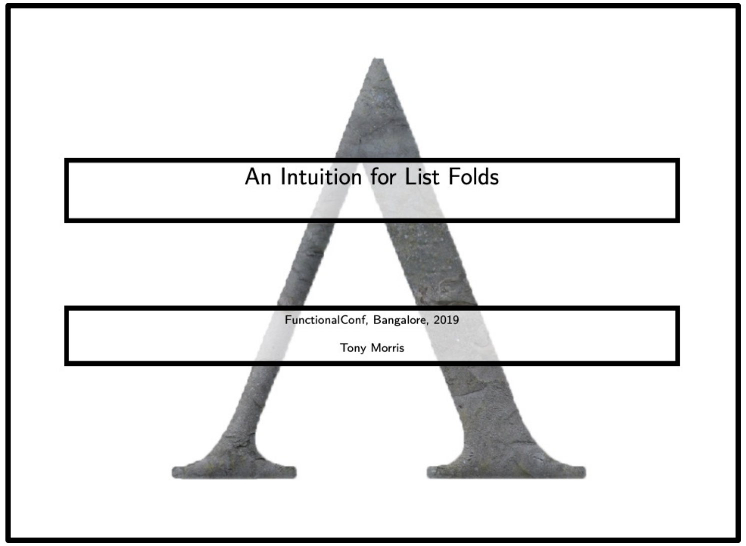
Polyglot FP for Fun and Profit
Haskell and Scala

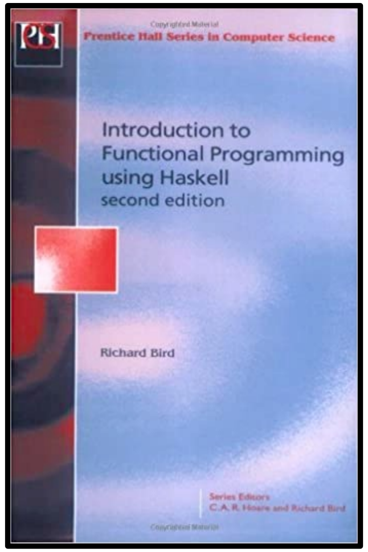
Develop the **correct intuitions** of what **fold left** and **fold right** actually do, and how different these two functions are

Learn other important concepts about **folding**, thus reinforcing and expanding on the material seen in parts 1 and 2

Includes a brief introduction to (or refresher of) **asymptotic analysis** and **Θ-notation**

Part 3 - through the work of





Tony Morris
@dibblego

YouTube https://presentations.tmorris.net/

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

slides by

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

@philip_schwarz

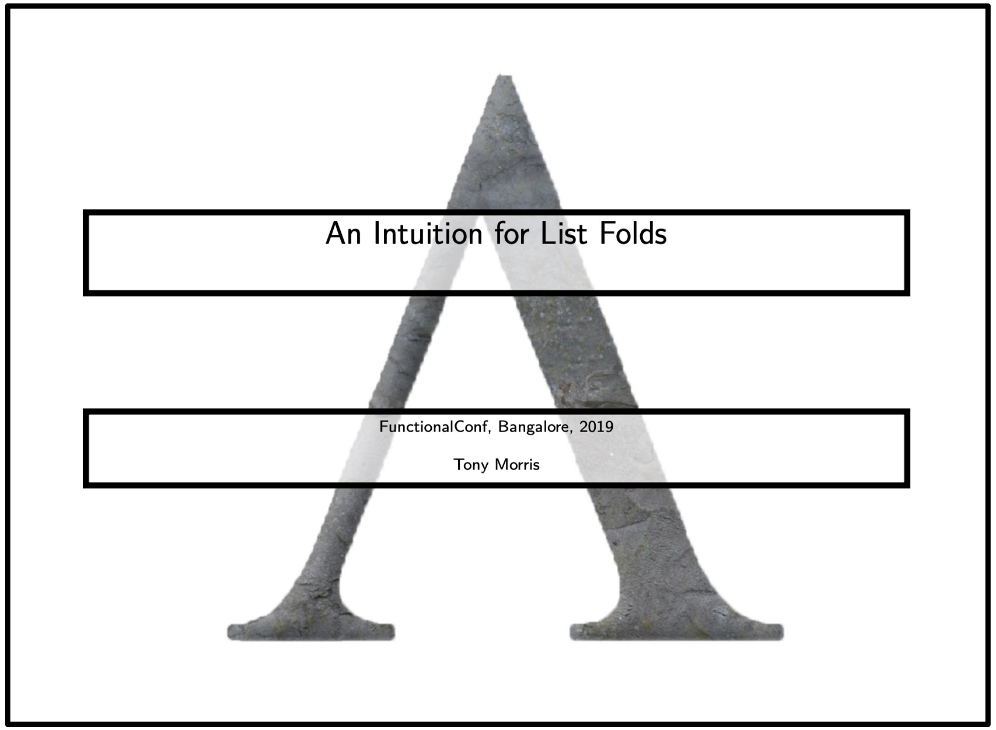
In this part of the series we are going to go through what I think is a very useful talk by **Tony Morris**.

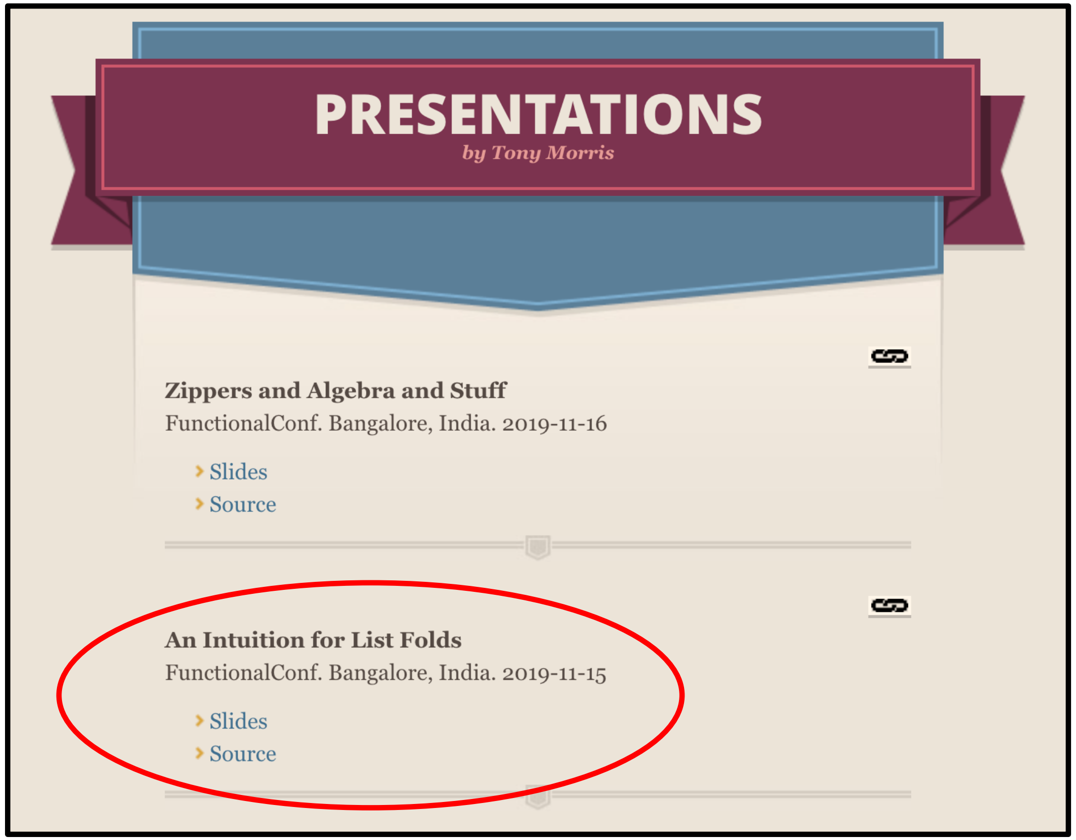
While it is a beginner level talk, IMHO **Tony** does a great job of explaining a number of important concepts about **folding**, including the **correct intuitions** to have about what **fold left** and **fold right** actually do, and how different these two functions are.

And as usual, we'll be looking for opportunities to expand on some topics and making a number of other interesting observations, allowing us to reinforce and expand on what we have already learnt in Parts 1 and 2.

Tony Morris

@dibblego





https://presentations.tmorris.net/

Tony Morris
@dibblego

Hello, my name is **Tony Morris**.

I am going to talk to you today about **list folds**.

It's a beginner level talk. I am hoping to transfer some knowledge to you to think abut **list folds** so that you can really understand how they work. …

OK so what are the goals for today?

I have heard of these **folds**… **left** and **right**

- What do they do?
- How do I know when to use them?
- Which one do I use?
- Can I internalize how they work?

Who has heard of **left** and **right fold on lists**? And for those of you who have your hand up, is that the end of your knowledge? That's it, you just heard of them? You have heard of them but that's it. A few people.

My goal today is to transfer you some knowledge so that you can **understand internally what they do**.

I get a lot of questions about them in my email. Can you tell me **when to use the right one**? What does this one do? What does that one do? **How do I think about them**?

I want to answer these questions.

Tony Morris
@dibblego

First we have to talk about what exactly is a **list**.

What is a **list**?

a list is either

- a *Nil* construction, with no associated data
- A *Cons* construction, associated with one arbitrary value, and another list

And **never**, **ever** anything else

A **list** is either *Nil*, an **empty list**, it carries no information, it is just an **empty list**.

Or, it has one element, and then another **list**.

Think about **lists** this way. I can make any **list** this way.

Using either *Nil* or *Cons*. *Nil* being an **empty list**. *Cons* having one element and then another **list**.

It is never anything else. It is always *Nil* or *Cons*.

Tony Morris
@dibblego

So this is the Haskell signature for them:

A list that holds elements of type a is constructed by either:

$$Nil :: List\ a$$
$$Cons :: a \to List\ a \to List\ a$$

So we say that Nil is just a $List$ of elements a, it's the **empty list**.

And $Cons$ takes an a, the first element, and then a $List\ a$, the rest of the **list**, and it makes a new **list**.

The word $Cons$ by the way goes back to the 1950s. We tend not to make up new words when they are that well established.

Here is the Haskell source code:

A list declaration using Haskell

$$\textbf{data}\ List\ a\ = Nil \mid Cons\ a\ (List\ a)$$

What this says is we are declaring a data type called $List$, carrying elements of type a. It is made with Nil, that has nothing, or with $Cons$, that has an a and another $List$ of a.

Tony Morris

@dibblego

How can we make **lists** using this?

For example, here is a **list** that has one element, the number 12. I have called ***Cons***, I passed in one element, 12, and then the rest of the **list**, ***Nil***, there is no rest of the **list**.

```
Haskell

Cons 12 Nil

printed

[12]
```

What about the **list** abc? I call ***Cons***, I pass in the letter ‘a’, then I have to pass in another **list**, so then I call ***Cons***, and the letter ‘b’, need to pass in another list, ***Cons***, ‘c’, ***Nil***.

```
Haskell

Cons ‘a’ (Cons ‘b’ (Cons ‘c’ Nil))

printed

[‘a’, ’b’, ’c’]
```

I can make any **list** using ***Cons*** and ***Nil***. That’s the definition of a **list**, or a ***Cons*** **list** as they are sometimes known.

Tony Morris

@dibblego

Sometimes you'll see *Nil* spelt square brackets. It's the same thing.

Naming conventions

- sometimes you will see *Nil* denoted [ ]
- and *Cons* denoted **:** which is used in infix position
- like this 1 **:**(2 **:**(3 **:**[ ]))
- but this is the same data structure

Sometimes you'll see *Cons* as just a colon, or sometimes a double colon, depending on the language.

So here is the **list** 1-2-3: one, *Cons*, and then a whole new **list**, 2, *Cons*, and then a whole new **list**, 3, *Cons* and then *Nil*.

This is the definition of a **list**. This is how we make them.

So when we talk about fold, we talk about these kinds of lists.

Footnote: there are languages for which this is not true. They talk about other kinds of **lists**. But if we consider **C# for example, it has an aggregate function which is a kind of fold, but it works on other kinds of lists, so it is not really a fold**.

So I am just going to talk about it in terms of *Cons* **lists**.

Tony Morris
@dibblego

Nearly two thirds of you have put your hand up, you have heard about **left fold** and **right fold**. Heard of them, that's it. Walking down the street one day, someone said "**left** and **right fold**", and then you just kept walking.

Left, Right, FileNotFound

- you may have heard of ***right folds*** and ***left folds***
- **Haskell**: **`foldr`**, **`foldl`**
- **Scala**: **`foldRight`**, **`foldLeft`**
- **C#** (BCL): no **right fold**, **`Aggregate`** (kind of)

In **Haskell** they are called **`foldr`** and **`foldl`**. In **Scala** they are called **`foldRight`** and **`foldLeft`**. And **C#** has this function called **`Aggregate`**, which is essentially a **`foldLeft`** (kind of).

Developing intuition for folds

- When do I know to use a **fold**?
- When do I know which **fold** to use?
- What do the **fold** functions *actually* do?

Just to be clear on our goals, when do I know to use a **fold**? What problem do I have so that I am going to use a **fold**? Which one am I going to use? And finally, what do they do? What is a good way to think about what they do?

Tony Morris

@dibblego

You might have seen these diagrams, they are on the internet. They are pretty good diagrams. They are quite accurate. They don't really help I think, in my experience.

There is much effort toward answering these questions

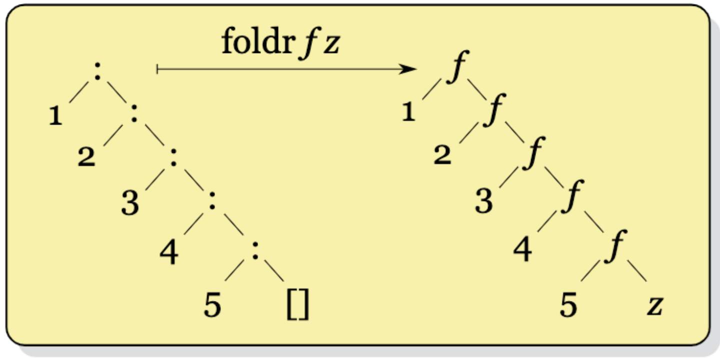


Figure: **right fold** diagram

People come up to me and say: **can you tell me exactly what a right fold is**? And I show them this diagram. And they go: **I still don't know what a right fold does**. It needs some explanation.

Tony Morris

@dibblego

This is a **left fold** diagram: it didn't help.

There is much effort toward answering these questions

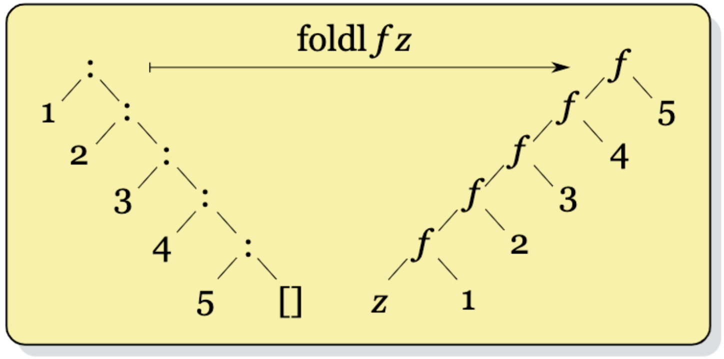


Figure: **left fold** diagram

And sometimes you probably heard of this

and terse explanations

- the **right fold** does **folding** from the **right** and **left fold**, **folding** from the **left**
- choose the **right fold** when you need to work with an **infinite list**

The **right fold** does **folding** from the **right** and **left fold** from the **left**. **Not only it is not helpful, it is not even true**.
I have also heard this: we are going to use the **right fold** when we need to work with an **infinite list**. **This is not correct**, OK?

Unfortunately

some of these explanations are incomplete or incorrect

Sometimes they are just not right.

Tony Morris

@dibblego

We seek an intuition that

- Does not require a prior **deep understanding** of **list folds**
- Goes far enough to leave us **satisfied**
- Is **not wrong**

We are looking for an intuition that doesn't require you to already have expert knowledge.

That is **satisfactory**, that you feel like you have **understood something**.

And that's **not wrong**.

Have you ever read a **monad tutorial** on the internet? You'll find that **they meet the first two goals**.

Consider **burritos**.

You don't need a **deep understanding** of **burritos**.

Burritos are **satisfactory**.

But **monads** are not **burritos**. Sorry, they are not.

I am hoping to achieve all three of these.

Tony Morris
@dibblego

First things first

In practice, the **foldl** and **foldr** functions are **very different**
So let us think about and discuss each separately.

The way to think about these two different functions is very **different**.

The intuition for each of them is quite **different**.

So I am going to be trying to talk about each **differently**.

Tony Morris
@dibblego

The `foldl` function accepts three values

```
1. f :: b -> a -> b
2. z :: b
3. list :: List a
```

to get back a value of type b

```
foldl :: (b -> a -> b) -> b -> List a -> b
B FoldLeft<A,B>(Func<B, A, B>, B, List<A>)
```

Let’s talk about what **foldleft** does.

It takes a **function** type **f**, **b** to the element type **a**, to **b**.

I takes another element **b**.

And then it takes a **list** that we are doing a **fold** on.

I also wrote the **C#** signature there, if you prefer to read that. I do not.

The $foldl$ signature we saw in part 1.

$$foldl :: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$$

Tony Morris

@dibblego

?

How does `foldl` take three values to that return value?

How does it take these three values to return a value? It does this **loop**:

All **left folds** are **loops**

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

Everyone's heard of a **loop**, right? They taught that back at **loop** school. I remember. First year undergrad: **loop** school.

So if we look at this **loop**. Who has written a **loop** like this before? Everyone has.

Tony Morris
@dibblego

All left folds are loops

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

The `foldl` function accepts three values

```
1.   f :: b -> a -> b
2.   z :: b
3.   list :: List a
```

to get back a value of type b

```
foldl :: (b -> a -> b) -> b -> List a -> b
```

And importantly, these (in **red**) are the three components of the **loop** that we get to change.

We get to pass in **a function, what to do on each iteration of the loop**. That's the **b** to **a** to **b** (`b -> a -> b`), the **f** there.

The **z** there is the **b**, so that's **what value to start the loop at**.

And finally **list**, **the thing that we are looping on, or foldlefting on**.

So let's look at some real code.

Refactor some loops

let's look at a real code example

In the next slide we are going to see a **plus operator enclosed in parentheses**. We have already seen $(+)$,$(-)$, $(\times)$, and $(\uparrow)$ in part 1, where we defined them to be **curried binary functions** and where their definitions made use of **infix operators** $+$, $-$, $\times$, and $\uparrow$.

$$
\begin{aligned}
(+) &:: Nat \rightarrow Nat \rightarrow Nat \\
m + Zero &= m \\
m + Succ\ n &= Succ\ (m + n)
\end{aligned}
$$

$$
\begin{aligned}
(-) &:: Nat \rightarrow Nat \rightarrow Nat \\
m - Zero &= m \\
Succ\ m - Succ\ n &= m - n
\end{aligned}
$$

$$
\begin{aligned}
(\times) &:: Nat \rightarrow Nat \rightarrow Nat \\
m \times Zero &= Zero \\
m \times Succ\ n &= (m \times n) + m
\end{aligned}
$$

$$
\begin{aligned}
(\uparrow) &:: Nat \rightarrow Nat \rightarrow Nat \\
m \uparrow Zero &= Succ\ Zero \\
m \uparrow Succ\ n &= (m \uparrow n) \times m
\end{aligned}
$$

Back then I thought the explanation below would have been superfluous, but in our current context, I think it is useful.

Enclosing an **operator** in parentheses converts it to a **curried prefix function** that can be applied to its arguments like any other function. For example,

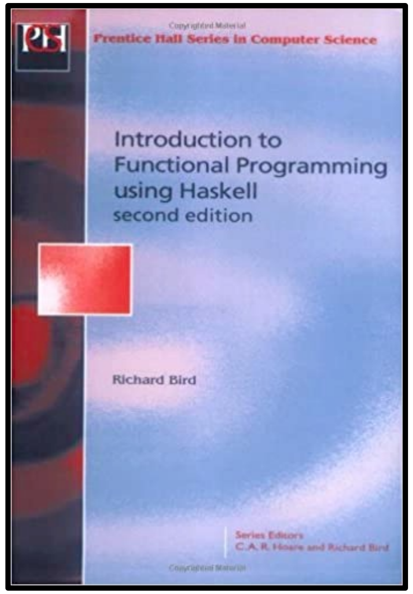
$$(+)\ 3\ 4 = 3 + 4$$
$$(\leq)\ 3\ 4 = 3 \leq 4$$

In particular,

$$plusc = (+)$$

where

$$
\begin{aligned}
plusc &:: Integer \rightarrow Integer \rightarrow Integer \\
plusc\ x\ y &= x + y
\end{aligned}
$$



Tony Morris
@dibblego

All left folds are loops

Let’s sum the integers of a list

Let’s add up the numbers in a **list**. Here is a **list** of numbers. Add them up.

What am I going to replace z with?

All left folds are loops

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

Well? **Zero**, yes. What about **f**? **Plus**? Yes, excellent. That will add up the numbers in the **list**.

sum the integers of a list

```
sum list = foldl (\r a -> (+) r a) 0 list
sum = foldl (+) 0
```

Left fold, given the accumulator through the **loop**, **r**, and the element **a**, add them, start the **loop** at **zero**, do it on the **list**.

This will add up the numbers in a **list**. And **if you eta-reduce that expression there, you end up with just plus**. **Just do plus on each iteration of the loop**.

On the previous slide, **Tony** just said the following: **if you eta-reduce that expression there, you end up with plus**.

η-reduction is one of the two forms of **η-conversion.**

η-conversion is adding or dropping of abstraction over a function. It converts between `λx.fx` and `f` (whenever `x` does not appear free in `f`).

η-expansion converts `f` to `λx.fx`, whereas **η-reduction** converts `λx.fx` to `f`.

Tony performed two consecutive reductions, one from `λx.λy.f x y` to `λx.f x`, and another from `λx.f x` to `f`.

In his case, `x` is called `r`, `y` is called `a`, `f` is `(+)`, and he reduced `λr.λa.(+) r a` to `(+)`.

"if you eta-reduce that expression there,…

```
sum list = foldl (\r a -> (+) r a) 0 list
```

η-reduction x 2

```
sum = foldl (+) 0
```

…you end up with plus"

@philip_schwarz

To help cement the notion of **eta-reduction** that we saw on the previous slide, and connect it to **Scala**, on this slide we do the following:

- compare the types of `(\r a -> (+) r a)` and `(+)` and see that they are the same
- show that `(\r a -> (+) r a)` and `(+)` behave the same

To also do that in **Scala**, we define the equivalent of **Haskell**'s `(+)` and **`foldl`** ourselves (see bottom of slide).

```
$ :type (\r a -> (+) r a)
(\r a -> (+) r a) :: Num a => a -> a -> a

$ :type (+)
(+) :: Num a => a -> a -> a

$ (\r a -> (+) r a) 3 4
=> 7

$ (+) 3 4
=> 7

$ foldl (\r a -> (+) r a) 0 [2,3,4]
=> 9

$ foldl (+) 0 [2,3,4]
=> 9
```

```
scala> :type (r:Int) => (a:Int) => `(+)`(r)(a)
Int => (Int => Int)

scala> :type `(+)`
Int => (Int => Int)

scala> ((r:Int) => (a:Int) => `(+)`(r)(a))(3)(4)
res1: Int = 7

scala> `(+)`(3)(4)
res2: Int = 7

scala> foldl((r:Int) => (a:Int) => `(+)`(r)(a))(0)(List(2,3,4))
res3: Int = 9

scala> foldl(`(+)`)(0)(List(2,3,4))
res4: Int = 9
```

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

scala>  val `(+)` = (x:Int) => (y:Int) => x + y
(+): Int => (Int => Int) = $$Lambda$5001/470155141@690b8d7f
```

Tony Morris
@dibblego

multiply the integers of a list

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

?

What about multiplication?

What do I replace the function **f** with? What are we going to do on each iteration of the **loop**?

We are going to do **multiplication**.

What are we going to start the **loop** at?

One. Some people say **zero**. What's going to happen if I put **zero** there? **Zero**. Yes.

One is the **identity for multiplication**. **One** is the thing that does nothing to **multiplication**. One times x gives me x. It did nothing to x.

Tony Morris

@dibblego

```
multiply the integers of a list

\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

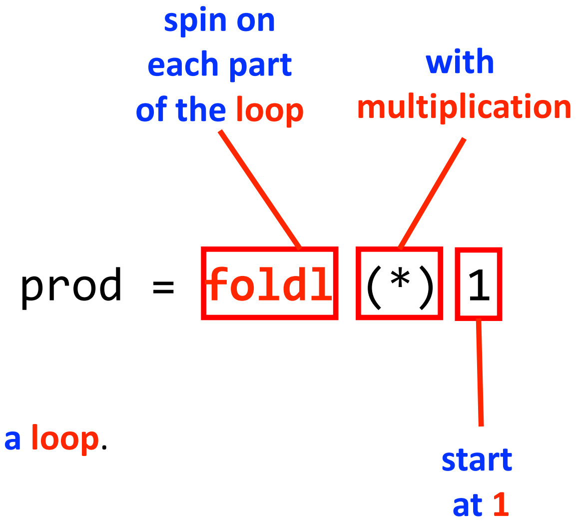
Replace the values in the loop

There it is. It's going to multiply the numbers in the **list**.

```
multiply the integers of a list

product list = foldl (\r a -> (*) r a) 1 list
product = foldl (*) 1
```

And there's the code. Real **Haskell** code. How to multiply the numbers in a **list**.



Left fold: **spin on each part of the loop with multiplication, start at 1. Fold left does a loop**.

all left folds are loops

I mean if you open up the source code of **fold left** you won't see a **loop** there. You'll see al sorts of crazy recursion and you'll see a seq or something like that to make it faster.

But **all you need to think about is it does a loop**, **that loop**.

Tony Morris

@dibblego

all left folds are loops

Let's reverse a list

How do you **reverse** a **list**? This was a trick question yesterday because I had taught everyone about **fold right**, and then I said ok, now **reverse** a **list**, and they tried to do it using **fold right**, and it ended up very slow.

Let's do it with a **left fold**.

reverse a list

```
\f z list ->
  var r = z
  foreach(a in list)
    r = f(r, a)
  return r
```

What am I going to replace **z** with, if I am going to **reverse** that **list**?

Nil, the **empty list**. And on each iteration of that **loop** I am going to take that element and put it on the front of that **list**.

That will reverse the **list**. **Left fold** through the **list**, pull the elements off the front and put them on the front of a new **list**, Nil, it will come back reversed, **in linear time**.

Tony Morris
@dibblego

```
reverse a list

\list ->
  var r = Nil
  foreach(a in list)
    r = flipCons(r, a)
  return r

flipCons = \r a -> Cons a r
```

There it is. I have a function. There is the **list** being accumulated **r**, there is the element of the **list a**, *Cons* it, do that in each iteration of the **loop**, start at *Nil*. This will **reverse** a **list**.

```
reverse a list

reverse list = foldl (\r a -> Cons a r) Nil list
reverse = foldl (flip Cons) Nil
```

That’s the real code.

I once went for a job interview, about twenty years ago, and the interviewer said to me, **reverse** a **list**. And I said, OK, what language. It was actually a **C#** job, and the guy said, any language you prefer. I said OK, **fold left** with *Cons Nil*. And I didn’t get the job. So I don’t recommend you answer that in that way. But it is correct. That will **reverse** a **list**.

Here is the definition of **reverse** that **Tony** showed us

```
reverse = foldl (flip Cons) Nil
```

We have already seen it in part 1

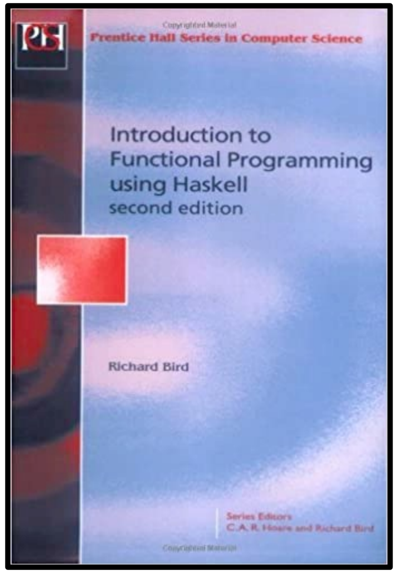
$$
\begin{aligned}
reverse' &:: [\alpha] \rightarrow [\alpha] \\
reverse' &= foldl\ cons\ [\,] \\
&\quad \textbf{where}\ cons\ xs\ x = x : xs
\end{aligned}
$$

Note the order of the arguments to $cons$; we have $cons = flip\ (:)$, where the standard function $flip$ is defined by $flip f\ x\ y = f\ y\ x$. The function $reverse'$, reverses a finite list.

Tony said that defining **reverse** using **foldr** ends up very slow, which we have also already seen in part 1

$$
\begin{aligned}
reverse &:: [\alpha] \rightarrow [\alpha] \\
reverse &= foldr\ snoc\ [\,] \\
&\quad \textbf{where}\ snoc\ x\ xs = xs +\!\!+ [x]
\end{aligned}
$$

$reverse'$ takes time proportional to n on a list of length n, while $reverse$ takes time proportional to n^2



Tony Morris
@dibblego

all left folds are loops

Let's compute the length of a list

What about the length of a **list**? What are we going to do? We are going to start the **loop** at **zero**, and for each of the **accumulators**, the **accumulator** **r**, we are going to ignore the element **a**, and just add one to **r**. That will compute the length of a **list**.

length of a list

```
\list ->
  var r = 0
  foreach(a in list)
    r = plus1(r, a)
  return r

plus1 = \r a -> r + 1
```

So, the function **plus1**, given **r**, ignore **a**, do **r + 1**, **do that on each spin of that loop**, it will compute the length of the **list**.

length of a list

```
length list = foldl (\r a -> r + 1) 0 list
length = foldl (const . (+ 1)) 0
```

There's the code. **I essentially read this word here (foldl) as do a loop. That's how I like to think about it. On each iteration of the loop, do that, start there**. That will compute the length of a **list**. This is just a point-free way of writing that same function. `const` means ignore the element, and then do **`plus1`**. On each iteration.

Tony Morris

@dibblego

refactoring, intuition

- a **left fold** is what you would write if I insisted you remove all duplication from your **loops**
- all **left folds** are exactly this **loop**
- any question we might ask about a **left fold**, can be asked about this **loop**.

If I said to you, **take all of the loops that you have written** and refactor out all of their differences, you'll end up with **fold left**. **They are exactly this loop**. That is to say, **I don't need a little footnote here to say,** "just kidding, it is not quite precise". **It is exactly that loop**. Which means that **any question we might ask about a left fold we can also ask about that loop, and we'll get the same answer**.

some observations

- a **left fold** will never work on an **infinite** list
- a correct intuition for **left folds** is easy to build on existing programming knowledge (**loop**).

For example, will that **loop** ever work on an **infinite list**? **Nope**. An **infinite list**, by the way, is one that doesn't have ***Nil***. It is just ***Cons*** all the way to **infinity**. If I put that into a **left fold** or into that **loop**, it just will never give me an answer. It will sit there and heat up the world a bit more.

It is easy to transfer this information because you probably have already heard of **loops**. I have used your existing knowledge to transfer this information. **Left fold** is a **loop**.

Folding to the left does a loop

sum the integers of a list

```
sum list = foldl (\r a -> (+) r a) 0 list
sum = foldl (+) 0
```

multiply the integers of a list

```
product list = foldl (\r a -> (*) r a) 1 list
product = foldl (*) 1
```

reverse a list

```
reverse list = foldl (\r a -> Cons a r) Nil list
reverse = foldl (flip Cons) Nil
```

length of a list

```
length list = foldl (\r a -> r + 1) 0 list
length = foldl (const . (+ 1)) 0
```

```
foldl :: (b -> a -> b) -> b -> List a -> b
foldl = \f z list ->
          var r = z
          foreach(a in list)
          r = f(r, a)
          return r
```

all **left folds** are **loops**

$$sum :: [\mathbf{Int}] \rightarrow \mathbf{Int}$$
$$sum = foldl\ (+)\ 0$$

On the left are **Tony**'s function definitions, and on the right are the definitions we saw in parts 1 and 2.

@philip_schwarz

$$prod :: [\mathbf{Int}] \rightarrow \mathbf{Int}$$
$$prod = foldl\ (\times)\ 1$$

$$reverse :: [\alpha] \rightarrow [\alpha]$$
$$reverse = foldl\ cons\ [\,]$$
$$\quad \mathbf{where}\ cons\ xs\ x = x : xs$$

$$length :: [\alpha] \rightarrow \mathbf{Int}$$
$$length = foldl\ plusone\ 0,$$
$$\quad \mathbf{where}\ plusone\ n\ x = n + 1$$

foldl can be seen as a loop because it is a **tail-recursive** function.

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

Tony Morris
@dibblego

Folding to the left does a loop

Folding to the left does a loop. The end.

For right folds there is no existing thing that I can use to transfer the information, you just simply need to commit to the definition of a list, which is, *Nil* or *Cons*. So let's commit to that right now. That's what a list is.

The fold right function.

The `foldr` function accepts three values

```
1. f :: a -> b -> b
2. z :: b
3. list :: List a
```

to get back a value of type b

```
foldr :: (a -> b -> b) -> b -> List a -> b
B FoldRight<A,B>(Func<A, B, B>, B, List<A>)
```

The `foldl` function accepts three values

```
1. f :: b -> a -> b
2. z :: b
3. list :: List a
```

to get back a value of type b

```
foldl :: (b -> a -> b) -> b -> List a -> b
B FoldLeft<A,B>(Func<B, A, B>, B, List<A>)
```

Well, it takes a function, a to b to b (a is the element type in the list), and then it takes a b, and it takes a list, and it returns a b. There it is, written in Haskell. There is it written in, Java, I think, I don't know. One of those languages.

What does it do? How does it take that function, that b, and that list and give me a b?

?
How does `foldr` take three values to that return value?

Tony Morris
@dibblego

constructor replacement

The `foldr` function performs **constructor replacement**.

The expression `foldr f z list` replaces in `list`:

- Every occurrence of ***Cons*** `(:)` with `f`.
- Any occurrence of ***Nil*** `[]` with `z`[1].

[1] The ***Nil*** constructor may be absent – i.e. the list is an **infinite** list of ***Cons***.

It performs **constructor replacement**. So, constructors, remember, are ***Nil*** and ***Cons***, they are the two things that construct **lists**. The expression **fold right** with the function **f**, **z** on a list, will go through that **list**, in no particular order, and replace every ***Cons*** with **f**, and ***Nil*** with **z**. If it sees a ***Nil***, which it might not, because it might be **infinite**.

constructor replacement?

- Suppose `list =` ***Cons*** `A (`***Cons*** `B (`***Cons*** `C (`***Cons*** `D` ***Nil***`)))`
- The expression `foldr f z list`
- produces `f A (f B (f C (f D z)))`

So if we take this list A, B, C, D, and I **fold right** with **f** and **z** on that **list**, I'll get back whatever value is replacing ***Cons*** with **f** and ***Nil*** with **z**, whatever that is.

So if A, B, C and D are all numbers and we want to add them up, I can replace **f** with plus, and **z** with **zero**, and it will add them all up.

Here on the right is Tony's explanation that **foldr** does **constructor replacement**, and below are the explanations we came across in Part 1.

constructor replacement

The `foldr` function performs **constructor replacement**.

The expression `foldr f z list` replaces in `list`:

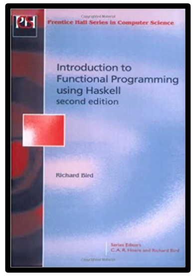
- Every occurrence of ***Cons*** `(:)` with `f`.
- Any occurrence of ***Nil*** `[]` with `z`[1].

[1] The ***Nil*** constructor may be absent – i.e. the list is an **infinite** list of ***Cons***.

Consider the following definition of a function h :

$$
\begin{aligned}
h\,[\,] &= e \\
h\,(x\!:xs) &= x \oplus h\,xs
\end{aligned}
$$

The function h works by **taking a list**, **replacing** $[\,]$ **by** e **and** $(:)$ **by** $\oplus$, **and evaluating the result**. For example, h converts the list



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

to the value

$$x_1 \oplus (x_2 \oplus (x_3 \oplus (x_4 \oplus e)))$$

Since $(:)$ associates to the right, there is no need to put in parentheses in the first expression. However, we do need to put in parentheses in the second expression because we do not assume that $\oplus$ associates to the right.

The pattern of definition given by h is captured in a function $foldr$ (pronounced 'fold right') defined as follows:

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

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

2 The fold operator

The **fold** operator has its origins in recursion theory (Kleene, 1952), while the use of **fold** as a central concept in a programming language dates back to the reduction operator of APL (Iverson, 1962), and later to the insertion operator of FP (Backus, 1978). In **Haskell**, the **fold** operator for lists can be defined as follows:

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

That is, given a function f of type $\alpha \to \beta \to \beta$ and a value v of type β, **the function** $fold\ f\ v$ **processes a list of type** $[\alpha]$ **to give a value of type** β **by replacing the nil constructor** $[\,]$ **at the end of the list by the value** v, **and each cons constructor** $(:)$ **within the list by the function** f. **In this manner, the** $fold$ **operator encapsulates a simple pattern of recursion for processing lists, in which the two constructors for lists are simply replaced by other values and functions**.

Tony Morris
@dibblego

multiply the integers of a list

Supposing
```
list = Cons 4 (Cons 5 (Cons 6 (Cons 7 Nil)))
```

?

Let’s multiply them. So here is a **list** of numbers, 4, 5, 6, 7. I am going to replace ***Cons*** with multiplication and ***Nil*** with one.

multiply the integers of a list

- let *Cons* = (*)
- let *Nil* = 1

And now, that will multiply the numbers in a **list**.

multiply the integers of a list

Supposing
```
list = (*) 4 ((*) 5 ((*) 6 ((*) 7 1)))

product list = foldr (*) 1 list
product = foldr (*) 1
```

Fold right did **constructor replacement**.

Tony Morris
@dibblego

right folds replace constructors

Let's and (&&) the booleans of a list.

The important thing about fold right to recognize, is that it doesn't do it in any particular order. There is an associativity order, but there is not an execution order. So that is to say, some people might say to me, fold right starts at the right side of the list. This can't be true, because I am going to be passing in an infinite list, which doesn't have a right side, and I am going to get an answer. If it started at the right, it went a really long way, and it is still going. So that is what I should see if that statement is true, but I don't see that. It associates to the right, it didn't start executing from the right. It's a subtle difference.

What if I have a **list** of booleans and I want to and them all up? What am I going to replace ***Nil*** with? Not 99. **True**. Yes.

and (&&) the booleans of a list
Supposing
```
list = Cons True (Cons True (Cons False (Cons True Nil)))
```

So if I have the above **list**, and I replace ***Nil*** with **True** and ***Cons*** with (&&), like this

and (&&) the booleans of a list

- let ***Cons*** = `(&&)`
- let ***Nil*** = `True`

It will and (&&) them all up

Tony Morris
@dibblego

and (&&) the booleans of a list
Supposing
```
list = (&&) True ((&&) True ((&&) False ((&&) True True)))

conjunct list = foldr (&&) True list
conjunct = foldr (&&) True
```

So there is the code. **Right fold** replacing *Cons* with (&&) and *Nil* with **True**. **It doesn't do it in any order.** I could have an infinite list of booleans. **Suppose I had an infinite list of booleans and it started at** **False**. *Cons* **False** something. **And I said** **foldr** (&&) **True**. **I should get back** **False**. **And I do. So clearly it didn't start from the right. It never went there. It just saw the** **False** **and stopped.**

How about appending two **lists**?

right folds replace constructors

Let's append two lists.

Here is a **list**. Here is a second **list**. How do I append them?

append two lists
Supposing
```
list1 = Cons A (Cons B (Cons C (Cons D Nil)))
list2 = Cons E (Cons F (Cons G (Cons H Nil)))
```

Do you agree with me that I am going to go through this first **list** and replace *Cons* with *Cons* and *Nil* with the second **list**? Who agrees with me on that? That's how you append two **lists**. Just an intuition for appending two **lists**. I take the first **list**, replace *Cons* with *Cons* and *Nil* with the other **list**, they are now appended.

Tony Morris
@dibblego

So now that you know that you should not be afraid when you see the code. I am going to go through this first **list** and replace ***Cons*** with ***Cons***, that is leave it alone, and I am going to pick up this entire `list2` and smash it straight over the ***Nil***. And that will be appended.

append two lists
- let ***Cons*** = ***Cons***
- let ***Nil*** = `list2`

So here is the code.

append two lists
Supposing

```
list1 = Cons A (Cons B (Cons C (Cons D Nil)))
list2 = Cons E (Cons F (Cons G (Cons H Nil)))
append list1 list2 = foldr Cons list2 list1
append = flip (foldr Cons)
```

Go in list1, replace ***Cons*** with ***Cons*** and ***Nil*** with `list2`. This will append `list1` and `list2`.

Sometimes I show people this code and they get scared. Wow, hang on, what is going on here? I am used to **loops** and things. That's how you append **lists**. Or go to the pointer at the end and update it to the other list, something crazy like that.

But if you get an intuition for **fold right**, which is doing **constructor replacement**, it is pretty straightforward, right? ***Cons*** with ***Cons*** and ***Nil*** with `list2`. Of course it is going to append the two **lists** (The second definition is just a **point-free** form).

You might choose to say that at your next job interview. Hey man, append two lists, ok, `flip (foldr Cons)`. Tell me how it goes.

@philip_schwarz

Here is **Tony**'s definition of the **append** function.

```
append list1 list2 = foldr Cons list2 list1
append = flip (foldr Cons)
```

We have already come across the function in part 1, where **Richard Bird** called it **concatenation**, and defined it **recursively**

$$(+\!\!+) \quad :: \quad [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$$
$$[\,]\; +\!\!+\; ys \quad = \quad ys$$
$$(x : xs)\; +\!\!+\; ys \quad = \quad x : (xs\; +\!\!+\; ys)$$

Concatenation takes two lists, both of the same type, and produces a third list, again of the same type.

```
def concatenate[A]: List[A] => List[A] => List[A] =
  xs => ys => xs match {
    case Nil => ys
    case x :: xs => x :: concatenate(xs)(ys)
  }
```

```
assert( concatenate(List(1,2,3))(List(4,5)) == List(1,2,3,4,5) )
```

Then in **TUEF** we saw the function defined in terms of $foldr$

$$(+\!\!+) \quad :: \quad [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$$
$$(+\!\!+\; ys) \quad = \quad foldr\ (:)\ ys$$

```
def concatenate[A]: List[A] => List[A] => List[A] = {
  def cons: A => List[A] => List[A] =
    x => xs => x :: xs
  xs => ys => foldr(cons)(ys)(xs)
}
```

Let's take Tony's two definitions of append, and translate them into Scala. Unlike the Scala concatenate function on the previous slide, which is repeated below, and which relies on the foldr definition to its right, Tony's definitions use *Cons*.

```
append list1 list2 = foldr Cons list2 list1
append = flip (foldr Cons)
```

```
def concatenate[A]: List[A] => List[A] => List[A] = {
  def cons: A => List[A] => List[A] =
    x => xs => x :: xs
  xs => ys => foldr(cons)(ys)(xs)
}
```

$$(+\!\!+) :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$$
$$(+\!\!+\ ys) = foldr\ (:)\ ys$$

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

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

So let's first modify the Scala version of *foldr* to use *Nil* and *Cons*

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

```
sealed trait List[+A]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
case object Nil extends List[Nothing]
```

We can now write the Scala equivalent of Tony's first definition of append

```
append list1 list2 = foldr Cons list2 list1
```

```
def append[A]: List[A] => List[A] => List[A] =
  xs => ys => foldr[A, List[A]]((Cons[A] _).curried)(ys)(xs)
```

NOTE: `(Cons[A] _)` has type `(A, List[A]) => List[A]`, whereas `((Cons[A] _).curried)` has type `A => List[A] => List[A]`

And if we write a Scala version of flip, we can then also translate into Scala Tony's second definition of append.

```
append = flip (foldr Cons)
```

```
def flip[A,B,C]: (A => B => C) => (B => A => C) =
  f => b => a => f(a)(b)

def append[A]: List[A] => List[A] => List[A] =
  flip(foldr((Cons[A] _).curried))
```

I don't know about you, but when I see **append** implemented so simply and elegantly in terms of **fold right**, I can't help wanting to see how **append** looks like when defined using **fold left**. The quickest way I can think of, for coming up with such a definition is to apply the **third duality theorem** of **fold**.

Here again is **Tony**'s definition of the **append** function.

```
append list1 list2 = foldr Cons list2 list1
```

Third duality theorem. For all finite lists xs,

$$foldr\ f\ e\ xs\ = foldl\ (flip\ f)\ e\ (reverse\ xs)$$
$$\textbf{where}\ flip\ f\ x\ y = f\ y\ x$$

And here again is the **third duality theorem**.

Let's use the theorem the other way round. Let's take the above definition of **append** in terms of **fold right**, and do the following:

- flip the first parameter of **fold right**
- reverse the third parameter of **fold right**
- replace **fold right** with **fold left**

```
append list1 list2 = foldl scon list2 (reverse list1)
                     where scon xs x = Cons x xs
```

Tony Morris
@dibblego

right folds replace constructors

Let's map a function on a list

What about **mapping** a function on a **list**? So who's heard of the **map** function? Or who's never heard of it? Everyone has. We have a **list**, and for each of the elements, I want to run a function on that element, to make a **new list**. Like I might have a **list** of numbers and I want to add ten to all of the numbers, I want to map + 10 on that **list**.

So here is my **list**

map a function (f) on a list

Supposing

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

?

What do I want to replace ***Cons*** with? Given the function **f**, do you agree that I want to say, ***Cons***, **f** of A, ***Cons***, **f** of B, ***Cons***, **f** of C, and D and then ***Nil***? That's what **map** does. I want to replace ***Cons*** with **f** and then ***Cons***. And ***Nil*** with ***Nil***.

map a function (f) on a list

- let *Cons* = `\x -> Cons (f x)`
- let *Nil* = *Nil*

So, given **x** I want to call **f**, then ***Cons***. And ***Nil*** with ***Nil***. This will **map** the function **f** on a **list**.

Tony Morris
@dibblego

map a function (f) on a list

Supposing

```
consf x = Cons (f x)
list = consf A (consf B (consf C (consf D Nil)))

map f list = foldr (\x -> Cons(f x)) Nil list
map f = foldr (Cons . f) Nil
```

So there is the code. It's not that scary now, is it? That's how you **map** a function on a **list**. We replace ***Cons*** with `(\x -> Cons(f x))`, and ***Nil*** with ***Nil***. We have mapped a function on a **list**.

Once I had to write mapping a function on a **list** in **Java**. This was 15 years ago. I didn't use **fold right**. This is just like, **footnote: caution**. **If you use fold right in Java, what's going to happen? Stack overflow. Yes, because fold right is recursive. For every element in the list, it's building up a stack frame**. So you can imagine my disappointment when I called **fold right** on the JVM, with a **list** of 10,000 numbers, or whatever it was, and it just said: **Stack overflow – have a nice day**. Because the JVM I used to use, this is a long time ago, was the IBM JVM.

It did tail-call optimisation, but it didn't optimise this one because it wasn't in tail position. And it didn't work on infinite lists either. I had to make it a heap **list**. So I am just letting you know, that all of this sounds great, but if you run out the door right now and say, 'I am going to do it in Java,' **caution**. The same is true for Python, C#, I have tried it: **Stack overflow**.

This little operator here, the dot, is **function composition**. It takes two functions and glues them together to make a new function. So I'll give you a bit of an intuition for **function composition**. I read it from right to left. Call **f** and then call ***Cons***. So wherever we are in the **list**, somewhere in a ***Cons*** cell, which means it has an element right next to it, call **f** on that element, and then do ***Cons***. And replace ***Nil*** with ***Nil***.

I wonder what would happen if you said that in a job interview. I should try that. Someone will say **map** a function on a **list** and they are waiting for me to say **for loop**, and I go, no no, **fold right**.

The reason why **Tony** experienced that **stack overflow** when calling **foldRight** with a large list is that by definition, **foldRight** is **recursive**, but not **tail-recursive** (unlike **foldleft**), whereas as we saw in Part 2, in **Scala**, in more recent years, the **foldRight** function of **List** has been redefined to take advantage of the **third duality theorem** of **fold**, i.e. it is now defined in terms of **foldLeft**, in that it first **reverses** the list that it is passed, and then does that same **loop** that **foldLeft** would do, except that there is no need to do any function flipping: the **loop** can just apply the given function as it stands.

So no more **stack overflows**.

github.com/scala/scala/blob/v2.13.3/src/library/scala/collection/immutable/List.scala

```scala
  final override def foldRight[B](z: B)(op: (A, B) => B): B = {
    var acc = z
    var these: List[A] = reverse
    while (!these.isEmpty) {
      acc = op(these.head, acc)
      these = these.tail
    }
    acc
  }
```

Third duality theorem. For all finite lists xs,

$$foldr\ f\ e\ xs\ = foldl\ (flip\ f)\ e\ (reverse\ xs)$$
$$\textbf{where}\ flip\ f\ x\ y = f\ y\ x$$

Tony Morris
@dibblego

right folds replace constructors

Let's flatten a list of lists

What about flattening a **list** of **lists**? So we have a **list**, and each element is itself a **list**, and we want to **flatten** it down. What am I going to replace ***Cons*** with? Any ideas? **append**, the function we just wrote. Go through each ***Cons*** and replace it with the function that **appends** two **lists**, and ***Nil*** with ***Nil***. That will **flatten** the **list** of **lists**.

flatten a list of lists

- let *Cons* = **append**
- let *Nil* = *Nil*

```
flatten list = foldr append Nil  list
flatten = foldr append Nil
```

There is the code. **fold right append *Nil***.

fold right does **constructor replacement**.

Tony's definition of flatten is the same as that of the concat function we saw in Part 1.

```
flatten :: [[a]]->[a]
flatten = foldr append Nil
```

$$concat :: [[\alpha]] \to [\alpha]$$
$$concat = foldr\ (+\!\!+)\ [\,]$$

For comparison, here is the other definition of concat that we saw in Part 1, the one that does not use $foldr$.

$$\begin{aligned} concat &:: [[\alpha]] \to [\alpha] \\ concat\,[\,] &= [\,] \\ concat\,(xs : xss) &= xs +\!\!+ concat\ xss \end{aligned}$$

Richard Bird says in his book that the above definition of concat is exactly what we would get from the definition $concat = foldr\ (+\!\!+)\ [\,]$ by eliminating the $foldr$.

And in Part 1 we saw Graham Hutton explain how the universal property of $foldr$ can be used to go from a function definition that doesn't use $foldr$ to a definition that does (and also to go the other way round).

$$\begin{aligned} g\ [\,] &= v \\ g\,(x : xs) &= f\ x\ (g\ xs) \end{aligned} \quad \Leftrightarrow \quad g = foldr\ f\ v$$

universal property of $foldr$

$$\begin{aligned} sum &:: [Int] \to Int \\ sum\ [\,] &= 0 \\ sum\ (x : xs) &= x + sum\ xs \end{aligned}$$

$$sum = fold\ (+)\ 0$$

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

$$map\ f = fold\ (\lambda x\ ys \to f\ x : ys)\ [\,]$$

@philip_schwarz

For what it is worth, on this slide I just want to show that it looks like in simple cases, like in the case of the **append** function, it seems possible, and easy enough, to **eliminate foldr** using some informal code transformations.

```
append xs ys = foldr (:) ys xs
```

```
foldr               :: (a -> b -> b) -> b -> [a] -> b
foldr f e []        = e
foldr f e (x:xs)    = f x (foldr f e xs)
```

replace f with (:)
replace e with ys

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

replace foldr (:) with append
swap append parameters

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

Now back to **Tony**'s definition of **flatten**, or as it was called in Part 1, **concat**.

```
flatten :: [[a]]->[a]
flatten = foldr append Nil
```

```
concat :: [[α]] → [α]
concat  = foldr (++) [ ]
```

As **Richard Bird** points out in his book, since ++ (i.e. **append**) is **associative** with **unit** [ ], thanks to the **first duality theorem** of **fold**, *concat* can also be defined using ***foldl***.

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

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

For all **finite** lists xs.

```
concat :: [[α]] → [α]
concat  = foldl (++) [ ]
```

Richard Bird also observes that **eliminating *foldl*** from the definition of *concat* leads to the following program.

```
concat′ :: [α] → [α]
concat′ xss  = accum [ ] xss

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

Similarly, if we eliminate ***foldl*** from the definition of *reverse′*

```
reverse′ :: [α] → [α]
reverse′  =  foldl cons [ ]
             where cons xs x = x : xs
```

We get this program

```
reverse′ :: [α] → [α]
reverse′ xs  = accum [ ] xs

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

So eliminating ***foldl*** leads to a **tail-recursive** function definition that uses an **accumulator**.

As **Sergei Winitzki** explained in Part 2, introducing an **accumulator** in order to achieve **tail recursion** is known as the **accumulator trick**.

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

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

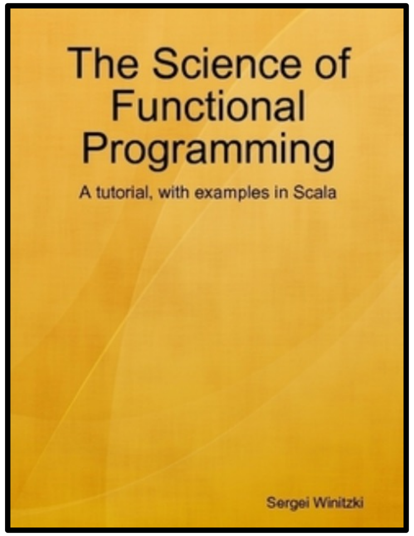
```
lengthT(Seq(1,2,3), 0)
 = lengthT(Seq(2,3), 1 + 0) // = lengthT(Seq(2,3), 1)
 = lengthT(Seq(3), 1 + 1)   // = lengthT(Seq(3), 2)
 = lengthT(Seq(), 1 + 2)    // = lengthT(Seq(), 3)
 = 3
```

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

sergei-winitzki-11a6431

```
append        :: [a] -> [a] -> [a]
append xs ys = foldl scon ys (reverse xs)
               where scon xs x = x : xs
```

```
foldl             :: (b -> a -> b) -> b -> [a] -> b
foldl f e []      = e
foldl f e (x:xs) = foldl f (f e x) xs
```

replace f with scon
replace e with ys

```
foldl scon ys []      = ys
foldl scon ys (x:xs) = foldl scon (scon ys x) xs
```

replace foldl scon with accum
inline remaining invocation of scon

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

define append in terms of accum

```
append xs ys = accum ys (reverse xs)
```

Again, for what it is worth, on this slide I just want to show that it looks like in simple cases, like in the case of the **append** function, it seems possible, and easy enough, to <u>eliminate</u> **foldl** using some informal code transformations.

@philip_schwarz

In both part 1 and in this part, we have come across the notion that sometimes it is more **efficient** to implement a function using a **right fold**, and at other times, it is more efficient to implement it using a **left fold**.

An effective way of comparing the **performance** of different definitions of a function is to carry out **asymptotic analysis** and then express the **performance** of each definition using the associated notation, i.e. O-notation, Ω–notation and Θ-notation.

The next four slides consist of a quick introduction to (refresher of) **asymptotic analysis**, and consists of extracts from **Richard Bird**’s book.

7.2 Asymptotic Analysis

In general, one is less interested in estimating the **cost** of evaluating a particular expression than in **comparing the performance of one definition of a function with another**. For example, consider the following two programs for reversing a list:

$$\begin{aligned} reverse\,[\,] &= [\,] \\ reverse\,(x : xs) &= reverse\ xs +\!\!+ [x] \end{aligned}$$

$$reverse' = foldl\ prefix\ [\,]\quad \textbf{where}\ prefix\ xs\ x = x : xs$$

It was claimed in section 4.5 that the second program is more **efficient** than the former, **taking at most a number of steps proportional to** n on a list of length n, while the first program **takes** n^2 **steps**. The aim of this section is to show how to make such claims more precise and to justify them.

7.2.1 Order notation

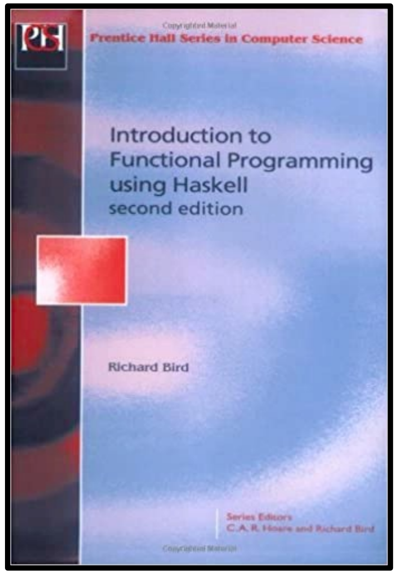
Given two functions f and g on the natural numbers, **we say that** f **is of order *at most*** g**, and write** $f = O(g)$ if there is a positive constant C and natural number n_0 such that $f(n) \leq Cg(n)$ for all $n \geq n_0$.

In other words, f **is bounded above by some constant times** g **for all sufficiently large arguments**.

The notation is abused to the extent that one conventionally writes, for example, $f(n) = O(n^2)$ rather than the more correct $f = O(square)$. Similarly, one writes $f(n) = O(n)$ rather than $f = O(id)$.

…

What O**-notation brings out is an upper bound on the *asymptotic* growth of functions**. For this reason, estimating the performance of a program using O-notation is called ***asymptotic upper-bound analysis***.



Richard Bird

For example, the **time complexity** of *reverse′* is $O(n)$. However, saying that *reverse* takes $O(n^2)$ steps on a list of length n does not mean that it does not take, say, $O(n)$ steps. For more precision we need additional notation.

We say that f is order *at least* g, and write $f = \Omega(g)$ if there exists a positive constant C and natural number n_0 such that $f(n) \geq Cg(n)$ for all $n \geq n_0$.

Putting the two kinds of bound together, **we say f is order *exactly* g, and write** $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$. In other words, $f = \Theta(g)$ if there are two positive constants C_1 and C_2 such that

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

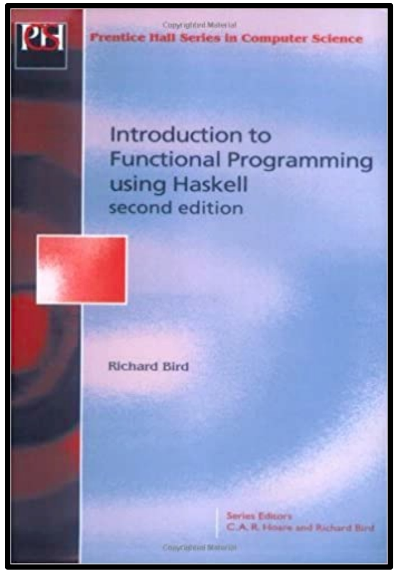
for all sufficiently large n. Then we can assert that **the time of *reverse*** is $\Theta(n^2)$ **and the time of *reverse′*** is $\Theta(n)$.

7.2.2 Timing analysis

Given a function f we will write $T(f)(n)$ to denote an asymptotic estimate of the number of reduction steps required to evaluate f on an argument of 'size' n in the worst case. Moreover, for reasons explained in a moment, **we will assume eager, not lazy, evaluation as a reduction strategy**. In particular, we can write

$$T(reverse)(n) = \Theta(n^2)$$
$$T(reverse')(n) = \Theta(n)$$

The definition of T requires some amplification. Firstly, $T(f)$ does not refer to the **time complexity** of a function f but to the **complexity** of a given ***definition*** of f. **Time complexity** is a property of an expression, not of the value of the expression.




Richard Bird

Secondly, we do not formalize the notion of **size**, since different measures are appropriate in different situations. For example, the cost of evaluating $xs +\!\!+\, ys$ is best measured in terms of m and n, where $m = length(xs)$ and $n = length(ys)$. In fact, we have

$$T(+\!\!+)(m, n) = \Theta(m)$$

The proof is left as an exercise. Next, consider $concat\ xss$. Here the measure of xss is more difficult. In the simple case that xss is a list of length m, consisting of lists of length n, we have

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

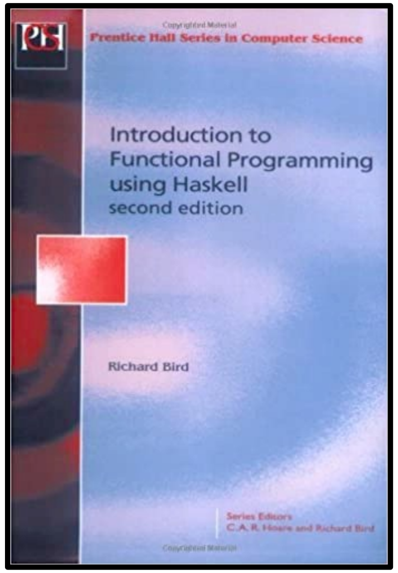
We will prove this result below. The estimate for $T(concat)$ therefore refers only to lists of lists with a common length; though limited, such restrictions make timing analyses more tractable.

The third remark is to emphasise that $T(f)(n)$ is an estimate of **worst-case** running time only. This will be sufficient for our purposes, although **best-case** and **average-case** analyses are also important in practice.

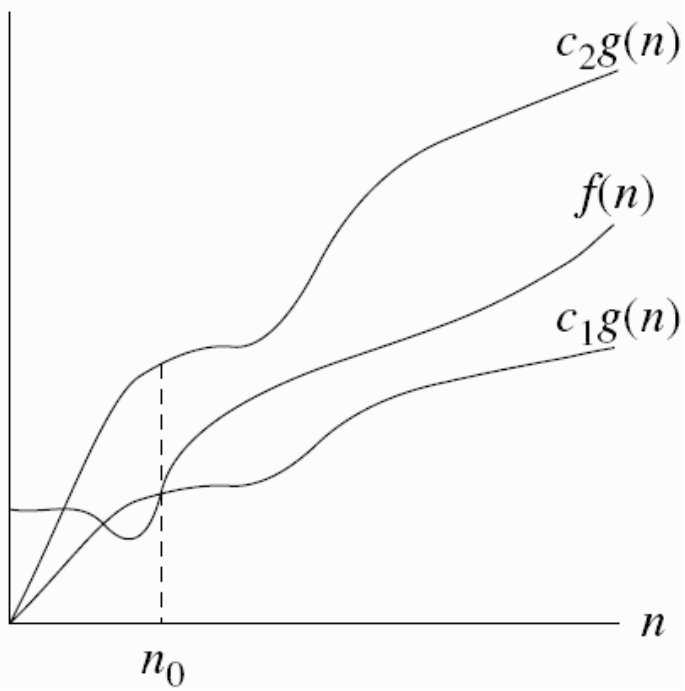
The fourth and crucial remark is that $T(f)(n)$ is determined under an **eager evaluation model** of **reduction**. The reason is simply that estimating the number of reduction steps under lazy evaluation is difficult, and is still the subject of ongoing research.

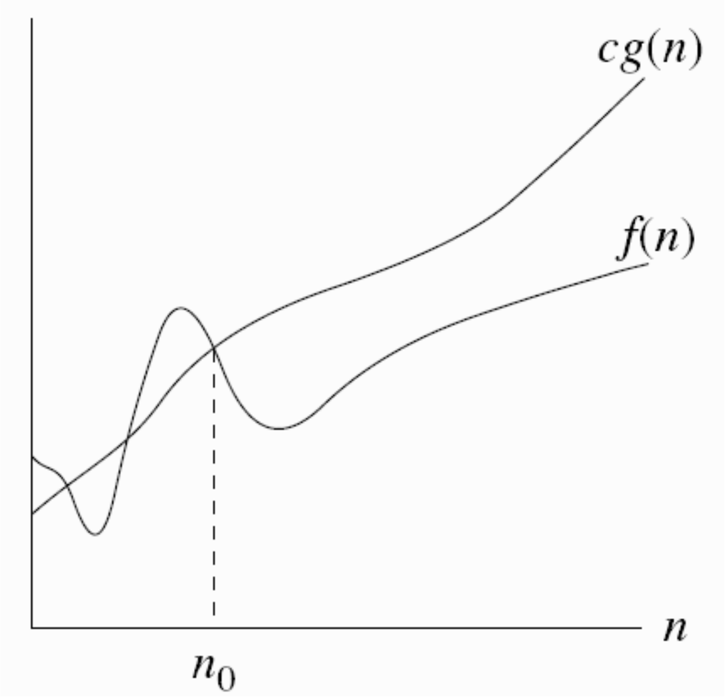
…

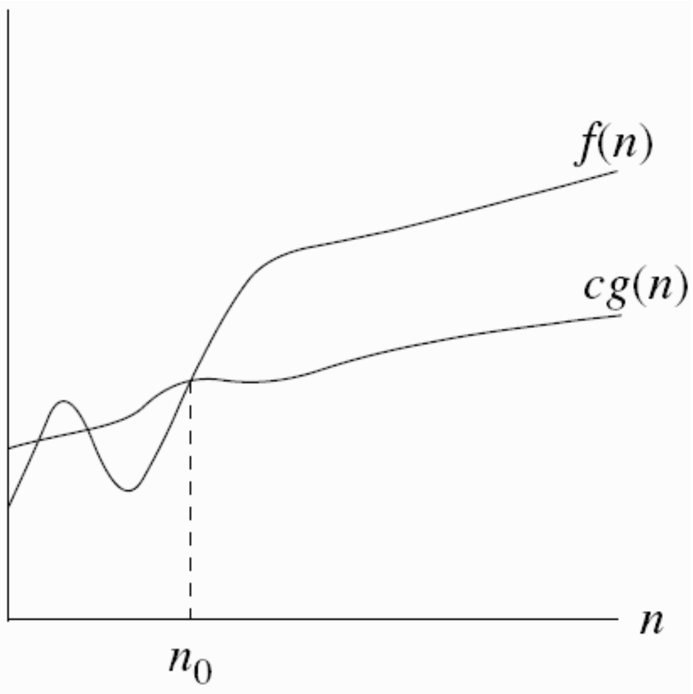
Timing analysis under **eager reduction** is simpler because it is **compositional**. Since **lazy evaluation** never requires more **reduction steps** than **eager evaluation**, any **upper bound** for $T(f)(n)$ will also be an **upper bound** under **lazy evaluation**. Furthermore, in many cases of interest, a **lower bound** for $T(f)(n)$ will also be a **lower bound** under **lazy evaluation**.



Richard Bird







$f(n) = \Theta(g(n))$

$C_1g(n) \leq f(n) \leq C_2g(n)$

for all sufficiently large n

$f(n) = O(g(n))$

$f(n) \leq Cg(n)$

for all $n \geq n_0$

$f(n) = \Omega(g(n))$

$f(n) \geq Cg(n)$

for all $n \geq n_0$

Images Source: *Introduction to Algorithms* (3rd edition)
by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein | Page 45 | Figure 3.1

Following that introduction to (refresher of) **asymptotic analysis**, this slide is a quick reminder, using Θ–notation, that whether it is more efficient to implement a function using *foldr*, or using *foldl*, depends on the function.

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

$T(reverse)(n) = \Theta(n^2)$

```
reverse′  :: [α] → [α]
reverse′  =  foldl scon [ ]
             where scon xs x = x : xs
```

$T(reverse')(n) = \Theta(n)$

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

$T(append)(m,n) = \Theta(m)$

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

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

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

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

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

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

I have renamed **cons** to **scon**, because I regard (:) as **cons**, and because the order of its arguments is the opposite of that of (:), and I find that the name **scon** conveys the fact that there is this inversion happening.

To be consistent with **Tony Morris**, we are defining **append** functions rather than an infix **append** operator ⧺.

I have added *xs* to the definition of **append**.

append′ is $\Theta(m)$ because in this case *foldl* is $\Theta(m)$, and *reverse′* is $\Theta(m)$.

That’s all for Part 3. I hope you found it useful.

We’ll continue looking at **Tony**’s presentation in Part 4.

See you there.