Fibonacci Function Gallery

Part 1

- Naïve Recursion
- Efficient Recursion with Tupling
- Tail Recursion with Accumulation
- Tail Recursion with Folding
- Stack-safe Recursion with Trampolining

Scala

Haskell

Clojure

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

@philip_schwarz

In this deck we are going to look at a number of different **implementations** of a **function** for computing the n^{th} element of the **Fibonacci sequence**.

To begin with, let's see how **Paul Hudak** introduces what is known as the **'naïve' implementation**.

#1 naïve implementation

14.2 Recursive Streams

Many problems are most easily solved using recursive streams. The use of recursive streams, a very powerful programming idiom, will be explored in detail in this section. Consider, for example, the Fibonacci sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

in which the first two numbers are 1, and each subsequent number is the sum of its two predecessors. The value of the *n*th Fibonacci number is defined mathematically as:

$$fib(n) = \begin{cases} 1 & if\ n = 0\ \vee n = 1 \\ fib(n-1) + fib(n-2) & if\ n \geq 2 \end{cases}$$

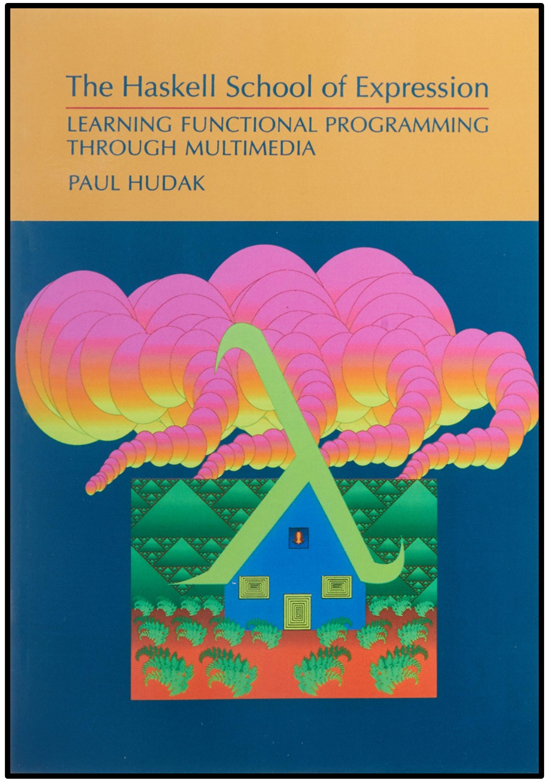
From this definition, a Haskell function can be defined straightforwardly to compute the nth Fibonacci number:

$$fib :: Integer \rightarrow Integer$$
$$fib\ 0 = 1$$
$$fib\ 1 = 1$$
$$fib\ n = fib\ (n-1) + fib\ (n-2)$$

There is only one problem: This function is horribly inefficient!

DETAILS

Try running this program on successively larger values of n; In Hugs, values larger than only 20 or so cause a noticeable delay.



Paul E. Hudak

#1 naïve implementation

To understand **the cause of this inefficiency**, let's begin the calculation of, say, $fib\ 8$:

$$fib\ 8$$

$$\Rightarrow fib\ 7 + fib\ 6$$

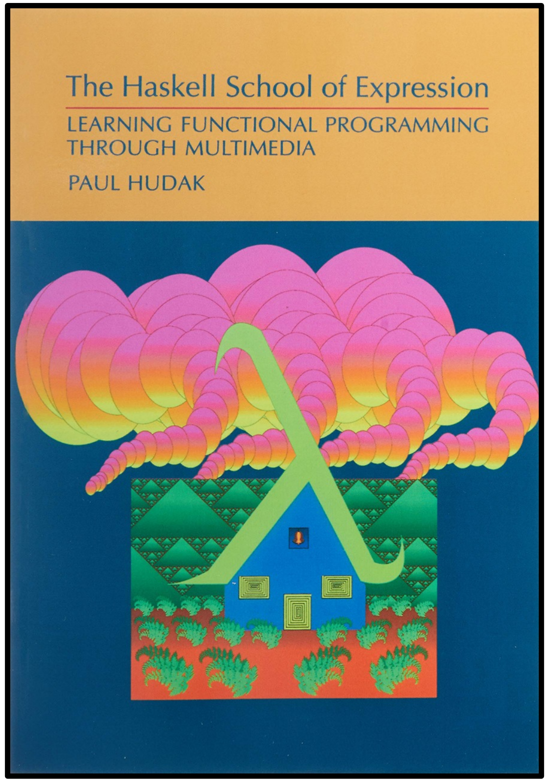
$$\Rightarrow (fib\ 6 + fib\ 5) + (fib\ 5 + fib\ 4)$$

$$\Rightarrow ((fib\ 5 + fib\ 4) + (fib\ 4 + fib\ 3)) + ((fib\ 4 + fib\ 3) + (fib\ 3 + fib\ 2))$$

$$\Rightarrow \left(\begin{array}{c}((fib\ 4 + fib\ 3) + (fib\ 3 + fib\ 2)) \\ + \\ ((fib\ 3 + fib\ 2) + (fib\ 2 + fib\ 1))\end{array}\right) + \left(\begin{array}{c}((fib\ 3 + fib\ 2) + (fib\ 2 + fib\ 1)) \\ + \\ ((fib\ 2 + fib\ 1) + (fib\ 1 + fib\ 0))\end{array}\right)$$

…

It is easy to see that **this calculation is blowing up exponentially**. That is, **to compute the nth Fibonacci number will require a number of steps proportional to 2^n. Sadly, many of the computations are being repeated**, but in general we cannot expect a **Haskell** implementation to realise this and take advantage of it. So what do we do?



Paul E. Hudak

#1 **naïve** implementation

As we have just seen, the **naïve[†] implementation** consists of a **recursive function** whose **time complexity** is **exponential** in its parameter.

Here is the **Scala** version of the **Haskell** function.

$$
\begin{aligned}
&fib :: Integer \rightarrow Integer \\
&fib\ 0 = 1 \\
&fib\ 1 = 1 \\
&fib\ n = fib\ (n-1) + fib\ (n-2)
\end{aligned}
$$

```scala
def fib(i: Int): BigInt = i match
  case 0 => 0
  case 1 => 1
  case _ => fib(i - 1) + fib(i - 2)
```

As for **recursive streams** (mentioned in the first of the previous two slides), we *will* be looking into their use later on.

† at this point we refer to the **naïve** implementation as such due to its **exponential time complexity**

As a minimal illustration of the **exponential time complexity** of the **naïve implementation**, here are a handful of very rough timings (on my laptop) for executing a program that just calls **fib** to compute the **nth factorial number**

`fib(35)`	about 3 seconds
`fib(40)`	about 5 seconds
`fib(45)`	about 14-15 seconds
`fib(50)`	about 2 minutes

In the next slide, **Richard Bird** first gives his explanation of why the **time complexity** of the **naïve implementation** is **exponential**, and then shows how the **tupling technique** can be used to produce a second **implementation** whose **time complexity** is **linear**.

7.4 Tupling

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

...

7.4.2 Fibonacci function

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

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

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

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

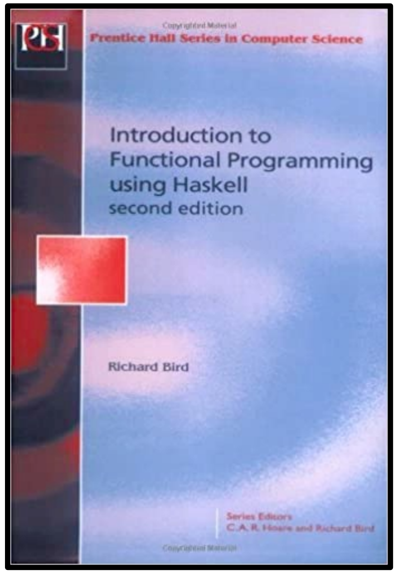
The **timing function** $T(fib)$ therefore satisfies equations very like that of fib itself. It is easy to check by **induction** that $T(fib)(n) = \Theta(fib\ n)$, so **the time to compute fib is proportional to the size of the result**. Since $fib(n) = \Theta(\phi^n)$, where ϕ is the **golden ratio**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

$$fibtwo\ n = (fib\ n, fib\ (n+1))$$

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

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

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



Richard Bird

#2 **tupling**-based implementation

As we have just seen, the second **implementation** also involves a **recursive function**, but its **time complexity** is **linear** (in its parameter), rather than **exponential**.

$$fib\ n = fst\ (fibtwo\ n)$$

$$fibtwo\ 0 = (0,1)$$
$$fibtwo\ (n+1) = (b, a+b), \quad \text{where } (a,b) = fibtwo\ n$$

Here is the **Scala** version of the **Haskell** function.

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) = i match
  case 0 => (0, 1)
  case _ => fibtwo(i - 1) match { case (fib_j, fib_k) => (fib_k, fib_j + fib_k) }
```

```scala
extension [A,B](pair: (A,B))
  def first: A = pair(0)
```

We can make it look a bit easier on the eye if we rename the recursively computed **fibonacci numbers** from fib_j, and fib_k to `a` and `b`.

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) = i match
  case 0 => (0, 1)
  case _ => fibtwo(i - 1) match { case (a, b) => (b, a + b) }
```

Remember these timings for the **naïve** implementation?

`fib(35)`	about 3 seconds
`fib(40)`	about 5 seconds
`fib(45)`	about 14-15 seconds
`fib(50)`	about 2 minutes

Contrast that with the fact that the **tupling**-based implementation takes only about 3-4 seconds to compute `fib(5,000)`.

In the next two slides, **Stuart Halloway** explains why the **naïve implementation** is **`stack-consuming`**.

```scala
def fib(i: Int): BigInt = i match
  case 0 => 0
  case 1 => 1
  case _ => fib(i - 1) + fib(i - 2)
```

#1 naïve implementation

Let's begin by implementing the **Fibonaccis** using a **simple recursion**. The following **Clojure** function will return the ***nth*** **Fibonacci** number:

```clojure
1: ; bad idea
2: (defn stack-consuming-fibo [n]
3:   (cond
4:     (= n 0) 0
5:     (= n 1) 1
6:     :else (+ (stack-consuming-fibo (- n 1))
7:              (stack-consuming-fibo (- n 2)))))
```

Clojure

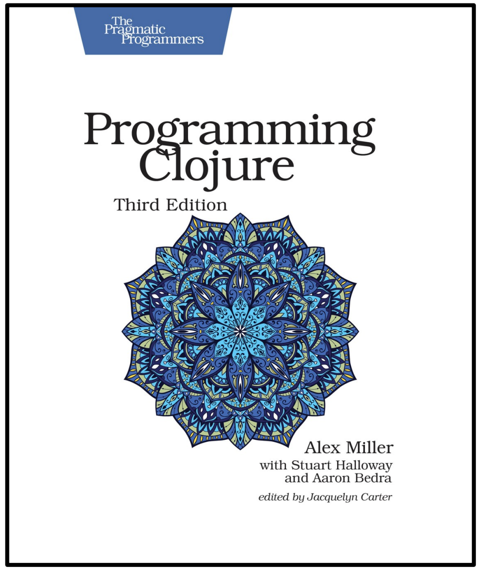
Lines 4 and 5 define the **basis**, and line 6 defines the **induction**. The implementation is **recursive** because **stack-consuming-fibo** calls itself on lines 6 and 7.

Test that **stack-consuming-fibo** works correctly for small values of ***n***:

```clojure
(stack-consuming-fibo 9)
-> 34N
```

Good so far, but **there's a problem calculating larger Fibonacci numbers such as F(1000000)**:

```clojure
(stack-consuming-fibo 1000000)
-> StackOverflowError clojure.lang.Numbers.minus (Numbers.java:1837)
```



Stuart Halloway
stuarthalloway

Because of the recursion, each call to stack-consuming-fibo for n > 1 begets two more calls to stack-consuming-fibo. At the JVM level, these calls are translated into method calls, each of which allocates a data structure called a *stack frame*.

The stack-consuming-fibo creates a depth of stack frames proportional to n, which quickly exhausts the JVM stack and causes the StackOverflowError shown earlier. (It also creates a total number of stack frames that's exponential in n, so its performance is terrible even when the stack does not overflow.)

Clojure function calls are designated as *stack-consuming* because they allocate stack frames that use up stack space. In Clojure, you should almost always avoid stack-consuming recursion as shown in stack-consuming-fibo.

Stuart Halloway

stuarthalloway

As an example of the naïve† implementation blowing the stack, if we try to use it to compute the ten thousandth Fibonacci number, we get a stack overflow error.

```
$ scala
Welcome to Scala 3.5.0 (22.0.2, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> def fib(i: Int): BigInt = i match
     |   case 0 => 0
     |   case 1 => 1
     |   case _ => fib(i - 1) + fib(i - 2)
     |
def fib(i: Int): BigInt


scala> fib(10_000)
java.lang.StackOverflowError
  at rs$line$1$.fib(rs$line$1:4)
  <…above line repeated 1023 more times…>
```

† at this point we refer to the naïve version as such due to both its exponential time complexity and its stack consumption

The **tupling**-based **implementation** is also **stack-consuming**, so it also **blows** the **stack** for sufficiently large **n**, e.g.

#2 **tupling**-based implementation

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) = i match
  case 0 => (0, 1)
  case _ => fibtwo(i - 1) match { case (a, b) => (b, a + b) }
```

```
scala> extension [A,B](pair: (A,B))
     |   def first: A = pair(0)
     |
def first[A, B](pair: (A, B)): A

scala> def fibtwo(i: Int): (BigInt, BigInt) = i match
     |   case 0 => (0, 1)
     |   case _ => fibtwo(i - 1) match { case (a, b) => (b, a + b) }
     |
def fibtwo(i: Int): (BigInt, BigInt)

scala> def fib(i: Int): BigInt =
     |   fibtwo(i).first
     |
def fib(i: Int): BigInt

scala> fib(7_500)
java.lang.StackOverflowError
  at rs$line$3$.fibtwo(rs$line$3:1)
 <…above line repeated 1023 more times…>
```

In the next 3 slides, **Stuart Halloway** looks at a **tail-recursive implementation**, and a **self-recursive implementation**.

Tail Recursion

Functional programs can solve the stack-usage problem with *tail recursion*. A tail-recursive function is still defined recursively, but the recursion must come at the tail, that is, at an expression that's a return value of the function. Languages can then perform tail-call optimization (TCO), converting tail recursions into iterations that don't consume the stack.

The **stack-consuming-fibo** definition of Fibonacci is not tail recursive, because it calls add (+) *after* both calls to **stack-consuming-fibo**. To make **fibo** tail recursive, you must create a function whose arguments carry enough information to move the induction forward, without any extra "after" work (like an addition) that would push the recursion out of the tail position. For **fibo**, such a function needs to know two Fibonacci numbers, plus an ordinal **n** that can count down to zero as new Fibonaccis are calculated. You can write **tail-fibo** as follows:

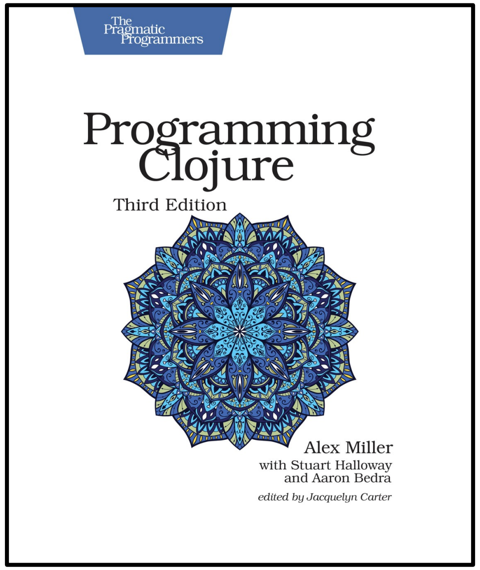
```
1: (defn tail-fibo [n]
2:   (letfn [(fib
3:              [current next n]
4:              (if (zero? n)
5:                current
6:                (fib next (+ current next) (dec n))))]
7:     (fib 0N 1N n)))
```

Line 2 introduces the `letfn` macro:

```
(letfn fnspecs & body)  ; fnspecs ==> [(fname [params*] exprs)+]
```

letfn is like **let** but is dedicated to creating local functions. Each function declared in a **letfn** can call itself or any other function in the same **letfn** block. Line 3 declares that **fib** has three arguments: the **current** Fibonacci, the **next** Fibonacci, and the number **n** of steps remaining.

Line 5 returns **current** when there are no steps remaining, and line 6 continues the calculation, decrementing the remaining steps by one. Finally, line 7 kicks off the **recursion** with the **basis values** 0 and 1, plus the ordinal **n** of the **Fibonacci** we're looking for.



Stuart Halloway
stuarthalloway

tail-fibo works for small values of **n**:

```
(tail-fibo 9)
-> 34N
```

But although it's tail recursive, it still fails for large **n**:

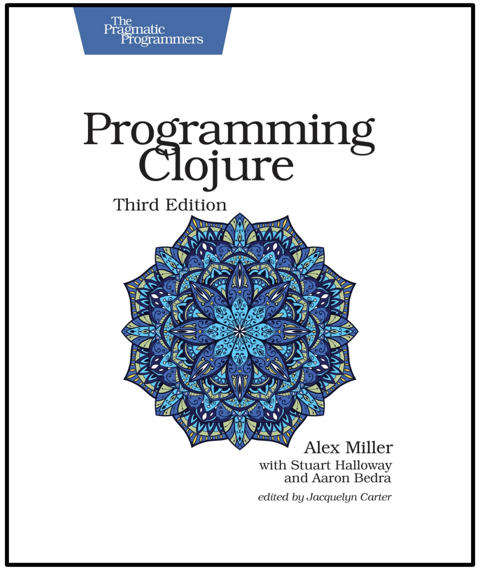
```
(tail-fibo 1000000)
-> StackOverflowError java.lang.Integer.numberOfLeadingZeros (Integer.java:1054)
```

The problem here is the JVM. While functional languages such as Scheme or Haskell perform TCO, the JVM doesn't perform this optimization. The absence of TCO is unfortunate but not a showstopper for functional programs.

Clojure provides several pragmatic workarounds: explicit self-recursion with **recur**, lazy sequences, and explicit mutual recursion with **trampoline**. We'll discuss the first two here and defer the discussion of **trampoline**, which is a more advanced feature, until later in the chapter.

Self-recursion with recur

One special (and common) case of recursion that *can* be optimized away on the JVM is self-recursion. Fortunately, the **tail-fibo** is an example: it calls itself directly, not through some series of intermediate functions.



Stuart Halloway
stuarthalloway

In Clojure, you can convert a function that tail-calls itself into an explicit self-recursion with **recur**. Using this approach, convert **tail-fibo** into **recur-fibo**:

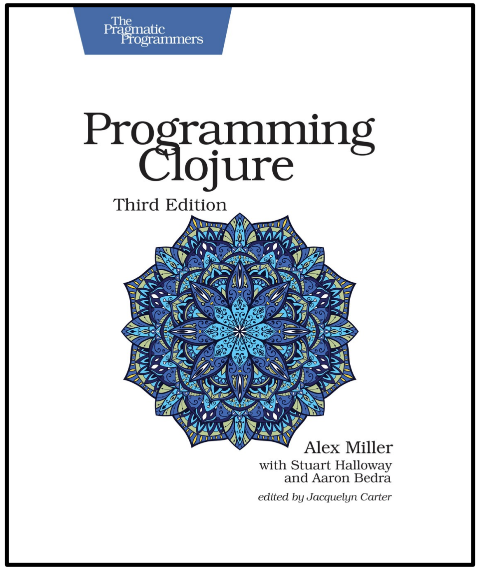
```
1:  ; better but not great
2:  (defn recur-fibo [n]
3:    (letfn [(fib
4:              [current next n]
5:              (if (zero? n)
6:                current
7:                (recur next (+ current next) (dec n))))]
8:      (fib 0N 1N n)))
```

The critical difference between **tail-fibo** and **recur-fibo** is on line 7, where **recur** replaces the call to **fib**.

The **recur-fibo** won't consume stack as it calculates Fibonacci numbers and can calculate F(n) for large **n** if you have the patience:

```
(recur-fibo 9)
-> 34N

(recur-fibo 1000000)
-> 195 ... 208,982 other digits ... 875N
```



Stuart Halloway
stuarthalloway

In **Scala** there is no need for **explicit self-recursion**: in the case of a function **recursively calling itself** in **tail position**, the compiler automatically performs **tail-call optimisation**.

See the next slide for how **Dean Wampler** puts it.

```scala
def fib(i: Int): BigInt = i match
  case 0 => 0
  case 1 => 1
  case _ => fib(i - 1) + fib(i - 2)
```

#1 naïve implementation

Recursion is a hallmark of FP and a powerful tool for writing elegant implementations of many algorithms. Hence, the Scala compiler does limited tail-call optimizations itself. It will handle functions that call themselves, but not *mutual recursion* (i.e., "a calls b calls a calls b," etc.).

Still, you might want to know if you got it right and the compiler did in fact perform the optimization. No one wants a blown stack in production. Fortunately, the compiler can tell you if you got it wrong if you add an annotation, tailrec, as shown in this refined version of factorial: …

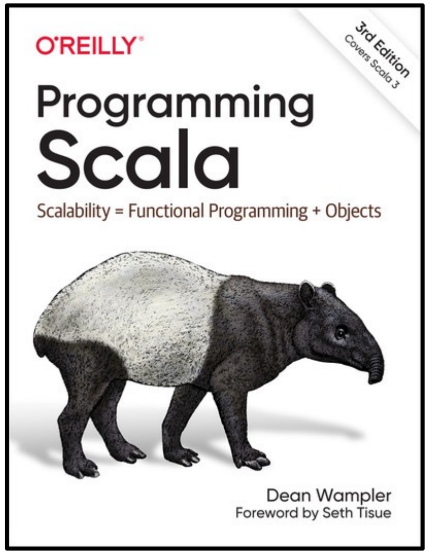
… If fact is not actually tail recursive, the compiler will throw an error. Consider this attempt to write a naïve recursive implementation of Fibonacci sequences:

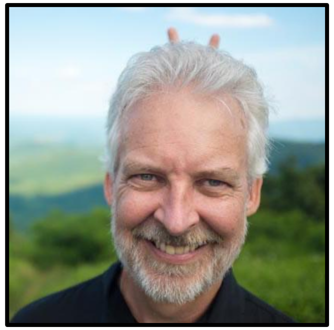
```
scala> import scala.annotation.tailrec

scala> @tailrec
     | def fibonacci(i: Int): BigInt =
     |   if i <= 1 then BigInt(1)
     |   else fibonacci(i - 2) + fibonacci(i - 1)
4 |   else fibonacci(i - 2) + fibonacci(i - 1)
  |                           ^^^^^^^^^^^^^^^^
  |                   Cannot rewrite recursive call: it is not in tail position
4 |   else fibonacci(i - 2) + fibonacci(i - 1)
  |        ^^^^^^^^^^^^^^^^
  |        Cannot rewrite recursive call: it is not in tail position
```

We are attempting to make two recursive calls, not one, and then do something with the returned values, in this case add them.

So this function is not tail recursive. (It is naïve because it is possible to write a tail recursive implementation.)





Dean Wampler
@deanwampler

#3 **tail-recursive** implementation

Here is the **Scala** version of the **Clojure** **tail-recursive**† **implementation**.

```clojure
(defn tail-fibo [n]
  (letfn [(fib
            [current next n]
            (if (zero? n)
              current
              (fib next (+ current next) (dec n))))]
    (fib 0N 1N n)))
```

```scala
def fib(i: Int): BigInt =
  tailFib(0, 1, i)

@tailrec
def tailFib(fib_j: BigInt, fib_k: BigInt, i: Int): BigInt = i match
  case 0 => fib_j
  case _ => tailFib(fib_k, fib_j + fib_k, i - 1)
```

Again, we can make it look a bit easier on the eye by renaming the recursively computed **fibonacci numbers** from fib_j and fib_k to `a` and `b`.

```scala
def fib(i: Int): BigInt =
  tailFib(0, 1, i)

@tailrec
def tailFib(a: BigInt, b: BigInt, i: Int): BigInt = i match
  case 0 => a
  case _ => tailFib(b, a + b, i - 1)
```

† the **tail-recursive** version is **not naïve** because it is **stack-safe** and its **time complexity** is **linear** rather than **exponential**

Remember the fact that the **tupling**-based implementation encountered a **stack overflow** when computing `fib(7,500)`?

As an example, the **tail-recursive** implementation is perfectly happy to compute `fib(10,000)`:

```
assert(fib(10_000) ==
BigInt("336447648764317832666216120051075433103021484606800639065647699746800814421666623681555955136337340255820653326808361593737347904838652682630408924630564318873545443695598274916066020998841839338646527313000888302692356736131351175792974378544137521305205043477016022647583189065278908551543661595829872796829875106312005754287834532155151038708182989697916131278562650331954871402142875326981879620469360978799003509623022910263681314931952756302278376284415403605844025721143349611800230912082870460889239623288354615057765832712525460935911282039252853934346209042452489294039017062338889910858410651831733604374707379085526317643257339937128719375877468974799263058370657428301616374089691784263786242128352581128205163702980893320999057079200643674262023897831114700540749984592503606335609338838319233867830561364353518921332797329081337326426526339897639227234078829281779535805709936910491754708089318410561463223382174656373212482263830921032977016480547262438423748624114530938122065649140327510866433945175121615265453613331113140424368548051067658434935238369596534280717687753283482343455573667197313927462736291082106792807847180353291311767789246590899386354593278945237776744061922403376386740040213303432974969020283281459334188268176838930720036347956231171031012919531697946076327375892535307725523759437884345040677155557790564504430166401194625809722167297586150269684431469520346149322911059706762432685159928347098912847067408620085871350162603120719031720860940812983215810772820763531866246112782455372085323653057759564300725177443150515396009051686032203491632226408852488524331580515348496224348482993809050704834824493274537326245677558790891871908036620580095947431500524025327097469953187707243768259074199396322659841474981936092852239450397071654431564213281576889080587831834049174345562705202235648464951961124602683139709750693826487066132645076650746115126775227486215986425307112984411826226610571635150692600298617049454250474913781151541399415506712562711971332527636319396069028956502882686083622410820505624307017949761711212330660733100599473668752"))
```

#4 **left fold**-based implementation

Next, here is an **implementation** using a **left fold**.

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  (1 to i).foldLeft(BigInt(0), BigInt(1))
    { case ((a, b), _) => (b, a + b) }
```

While it consists of two functions, neither of which is recursively defined, a **left fold** is **tail recursive**, so the **time complexity** of this **implementation** is **linear**.

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

```scala
@tailrec
def foldl[A,B](f: B => A => B)(b: B)(as: List[A]): B = as match
  case Nil   => b
  case x::xs => foldl(f)(f(b)(x))(xs)
```

Implementations explored so far

```scala
def fib(i: Int): BigInt = i match
  case 0 => 0
  case 1 => 1
  case _ => fib(i - 1) + fib(i - 2)
```

version #1 (**naïve**)
- not **tail-recursive** (not **stack-safe**)
- **exponential** time complexity
- **linear** stack frame depth

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) = i match
  case 0 => (0, 1)
  case _ => fibtwo(i - 1) match { case (a, b) => (b, a + b) }
```

version #2 (**tupling**-based)
- not **tail-recursive** (not **stack-safe**)
- **linear** time complexity
- **linear** stack frame depth

```scala
def fib(i: Int): BigInt =
  tailFib(0, 1, i)

@tailrec
def tailFib(a: BigInt, b: BigInt, i: Int): BigInt = i match
  case 0 => a
  case _ => tailFib(b, a + b, i - 1)
```

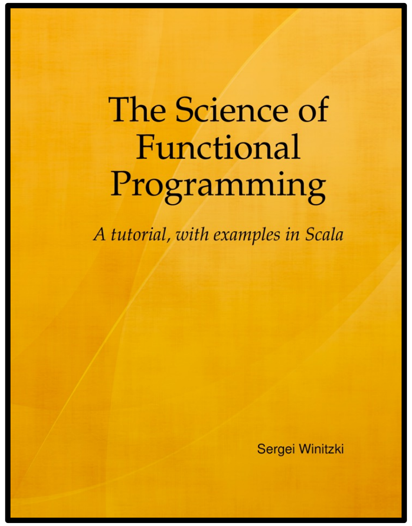
version #3 (**tail-recursive**)
- **tail-recursive** (**stack-safe**)
- **linear** time complexity

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  (1 to i).foldLeft(BigInt(0), BigInt(1))
    { case ((a, b), _) => (b, a + b) }
```

version #4 (**left fold**-based)
- **non-recursive** (**stack-safe**)
- **linear** time complexity

While we have already seen that it is possible to write a **tail recursive implementation** (e.g. **version #2**), there is a **technique**, called **trampolining**, that can be used to make even the **naïve implementation stack safe**.



There is **no automatic recipe** for converting an arbitrary function into a **tail-recursive** one.

The **accumulator trick** does not always work!

In some cases, it is impossible to implement **tail recursion** in a given recursive computation.

An example of such a computation is the “merge-sort” algorithm where the function body must contain **two recursive calls within a single expression**. (**It is impossible to rewrite two recursive calls as one tail call**.)

What if our recursive code cannot be transformed into **tail-recursive** code via the **accumulator trick**, but the **recursion depth** is so large that **stack overflows** occur?

There exist **special techniques** (e.g., “continuations” and **“trampolines”**) that convert **non-tail-recursive** code into code that runs without **stack overflows**.

Sergei Winitzki

sergei-winitzki-11a6431

In the next two slides we look at how **Noel Welsh** explains how the **Eval monad** in **Cats** can be used for **trampolining** purposes.

9.6.4 Trampolining and Eval.defer

One useful property of **Eval** is that its **map** and **flatMap** methods are **trampolined**.

This means we can **nest** calls to **map** and **flatMap** arbitrarily without consuming **stack frames**.

We call this property **"stack safety"**.

For example, consider this function for calculating **factorials**:

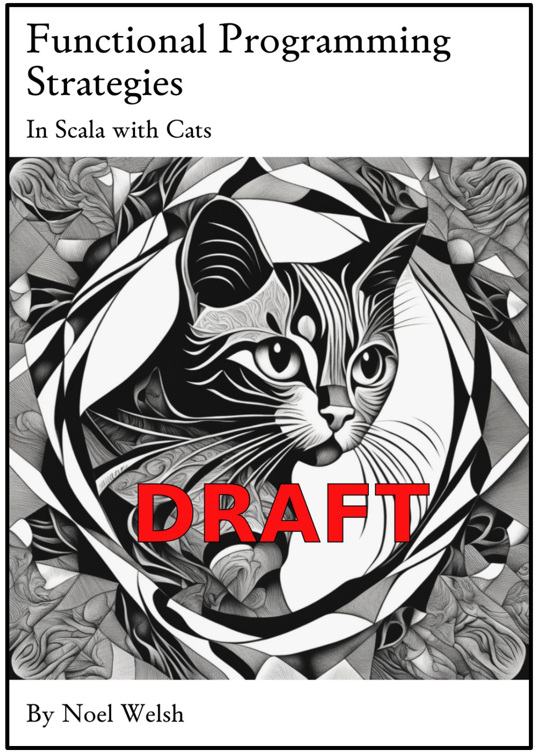
```scala
def factorial(n: BigInt): BigInt =
  if (n == 1) n else n * factorial(n - 1)
```

It is relatively easy to make this method **stack overflow**:

```scala
factorial(50000)
// java.lang.StackOverflowError
// ...
```

We can rewrite the method using **Eval** to make it **stack safe**:

```scala
def factorial(n: BigInt): Eval[BigInt] =
  if(n == 1) {
    Eval.now(n)
  } else {
    factorial(n - 1).map(_ * n)
  }
```



Noel Welsh

@noelwelsh

```scala
factorial(50000).value
// java.lang.StackOverflowError
// ...
```

Oops! That didn’t work—our **stack** still **blew up!**

This is because we’re still making all the **recursive calls** to **factorial** before we start working with **Eval**'s **map** method.

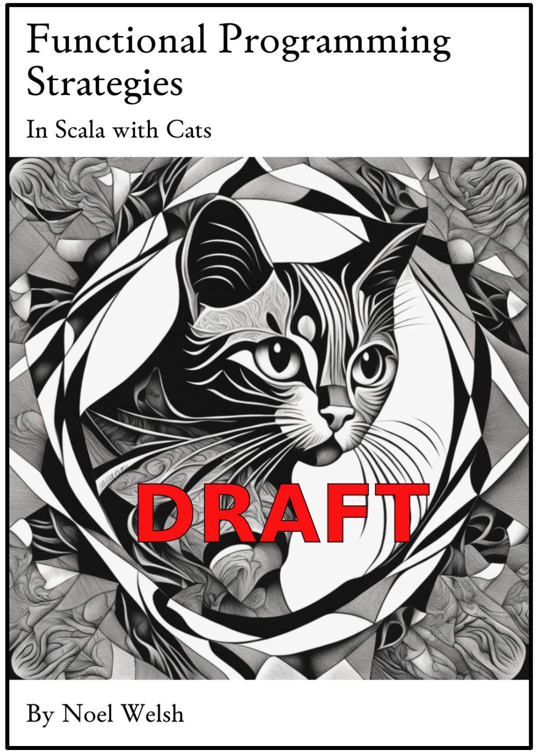
We can work around this using **Eval.defer**, which takes an existing instance of **Eval** and **defers** its **evaluation**.

The **defer** method is **trampolined** like **map** and **flatMap**, so we can use it as a quick way to make an existing operation stack safe:

```scala
def factorial(n: BigInt): Eval[BigInt] =
  if(n == 1) {
    Eval.now(n)
  } else {
    Eval.defer(factorial(n - 1).map(_ * n))
  }

factorial(50000)
// res: A very big value
```

Eval is a useful **tool** to **enforce stack safety** when working on **very large** computations and data structures. However, we must bear in mind that **trampolining** is **not free**. It avoids **consuming stack** by creating a **chain of function objects** on the **heap**. There are still **limits** on how **deeply** we can **nest** computations, but they are bounded by the size of the **heap** rather than the **stack**.



Noel Welsh

@noelwelsh

cats

Eval

```scala
/**
 * Eval is a monad which controls evaluation.
 *
 * This type wraps a value (or a computation that produces a value)
 * and can produce it on command via the `.value` method.
 *
 * There are three basic evaluation strategies:
 *
 *  - Now:    evaluated immediately
 *  - Later:  evaluated once when value is needed
 *  - Always: evaluated every time value is needed
 *
 * The Later and Always are both lazy strategies while Now is eager.
 * Later and Always are distinguished from each other only by
 * memoization: once evaluated Later will save the value to be returned
 * immediately if it is needed again. Always will run its computation
 * every time.
 *
 * Eval supports stack-safe lazy computation via the .map and .flatMap
 * methods, which use an internal trampoline to avoid stack overflows.
 * Computation done within .map and .flatMap is always done lazily,
 * even when applied to a Now instance.
 * …
 */
sealed abstract class Eval[+A] extends …
```

```scala
/**
 * Construct an eager Eval[A] value (i.e. Now[A]).
 */
def now[A](a: A): Eval[A] = Now(a)
```

```scala
def factorial(n: BigInt): Eval[BigInt] =
  if(n == 1) {
    Eval.now(n)
  } else {
    Eval.defer(factorial(n - 1).map(_ * n))
  }
```

```scala
/**
 * Defer a computation which produces an Eval[A] value.
 *
 * This is useful when you want to delay execution of an expression
 * which produces an Eval[A] value. Like .flatMap, it is stack-safe.
 */
def defer[A](a: => Eval[A]): Eval[A] = …
```

Here is how we can use the **Eval monad** to make the **naïve implementation stack-safe**

#5 stack-safe naïve Eval-based implementation

```scala
def fib(i: Int): BigInt = i match
  case 0 => 0
  case 1 => 1
  case _ => fib(i - 1) + fib(i - 2)
```

```scala
def fib(i: Int): Eval[BigInt] = i match
  case 0 => Eval.now(0)
  case 1 => Eval.now(1)
  case _ => Eval.defer(fib(i - 1))
                .flatMap(a => fib(i - 2)
                .map(b => a + b))
```

```scala
def fib(i: Int): Eval[BigInt] = i match
  case 0 => Eval.now(0)
  case 1 => Eval.now(1)
  case _ =>
    for
      a <- Eval.defer(fib(i - 1))
      b <- fib(i - 2)
    yield a + b
```

same as above, but using the **syntactic sugar** of a **for comprehension**

† we refer to this version as **naïve** because while it is **stack-safe**, its **time complexity** is still **exponential**

Remember these timings for the **naïve** implementation?

Naïve

`fib(35)` about 3 seconds
`fib(40)` about 5 seconds
`fib(45)` about 14-15 seconds
`fib(50)` about 2 minutes

While the **stack-safe naïve** implementation avoids **stack overflows**, it is much **slower** than the **naïve** one, e.g.

Stack-safe naïve

`fib(35)` about 5 seconds
`fib(40)` about 14-15 seconds
`fib(45)` about 2 minutes
`fib(50)` **?** I got bored of waiting after 20 minutes, but it had finished within 23.

If you are interested in knowing more about **trampolining**, consider taking a look at this

Game of Life - Polyglot FP
Haskell - Scala - Unison

Follow along as **Trampolining** is used to overcome **Stack Overflow** issues with the simple **IO monad**

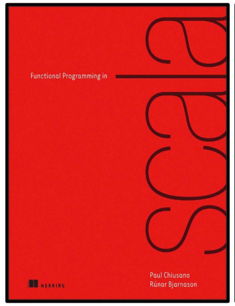
deepening you understanding of the **IO monad** in the process

See **Game of Life IO actions** migrated to the **Cats Effect IO monad**, which is **trampolined** in its **flatMap** evaluation

(Part 3)

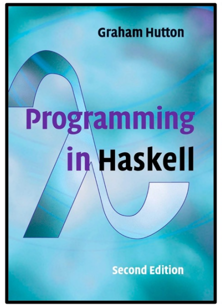
through the work of

Runar Bjarnason
@runarorama



FP in Scala

Paul Chiusano
@pchiusano



Graham Hutton
@haskellhutt

Cats Effect
The IO Monad for Cats

TYPELEVEL
SCALA

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

We can also do something similar with the **Cats Effect IO Monad**, because

1. its **map** and **flatMap** functions are also **trampolined**
2. it also provides a **defer** function.

By the way, the documentation for **IO** provides a **Fibonacci function** example!

cats.effect IO

```scala
/**
 * Lifts a pure value into `IO`.
 …
 */
def pure[A](value: A): IO[A] = …
```

…

IO is **trampolined** in its **flatMap** evaluation. This means that you can safely call **flatMap** in a **recursive function** of **arbitrary depth**, without fear of **blowing** the **stack**.

```scala
def fib(n: Int, a: Long = 0, b: Long = 1): IO[Long] =
  IO.pure(a + b) flatMap { b2 =>
    if (n > 0)
      fib(n - 1, b, b2)
    else
      IO.pure(a)
  }
```

```scala
/**
 * Suspends a synchronous side effect which produces an `IO` in `IO`.
 *
 * This is useful for trampolining (i.e. when the side effect is conceptually the allocation
 * of a stack frame). Any exceptions thrown by the side effect will be caught and sequenced
 * into the `IO`.
 */
def defer[A](thunk: => IO[A]): IO[A] = …
```

#6 stack-safe naïve IO-based implementation

Here is the IO equivalent of the Eval-based stack-safe implementation.

```scala
def fib(i: Int): Eval[BigInt] = i match
  case 0 => Eval.now(0)
  case 1 => Eval.now(1)
  case _ =>
    for
      a <- Eval.defer(fib(i - 1))
      b <- fib(i - 2)
    yield a + b
```

```scala
def fib(i: Int): IO[BigInt] = i match
  case 0 => IO.pure(0)
  case 1 => IO.pure(1)
  case _ =>
    for
      a <- IO.defer(fib(i - 1))
      b <- fib(i - 2)
    yield a + b
```

† we also refer to this version as naïve because while it is stack-safe, its time complexity is still exponential

To conclude part 1, the next slide is a recap of the different implementations that we have explored.

Implementations explored so far

1

```scala
def fib(i: Int): BigInt = i match
  case 0 => 0
  case 1 => 1
  case _ => fib(i - 1) + fib(i - 2)
```

2

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) = i match
  case 0 => (0, 1)
  case _ => fibtwo(i - 1) match
    { case (a, b) => (b, a + b) }
```

3

```scala
def fib(i: Int): BigInt =
  tailFib(0, 1, i)

def tailFib(a: BigInt, b: BigInt, i: Int): BigInt =
  i match
    case 0 => a
    case _ => tailFib(b, a + b, i - 1)
```

4

```scala
def fib(i: Int): BigInt =
  fibtwo(i).first

def fibtwo(i: Int): (BigInt, BigInt) =
  (1 to i).foldLeft(BigInt(0), BigInt(1))
    { case ((a, b), _) => (b, a + b) }
```

5

```scala
def fib(i: Int): Eval[BigInt] = i match
  case 0 => Eval.now(0)
  case 1 => Eval.now(1)
  case _ =>
    for
      a <- Eval.defer(fib(i - 1))
      b <- fib(i - 2)
    yield a + b
```

6

```scala
def fib(i: Int): IO[BigInt] = i match
  case 0 => IO.pure(0)
  case 1 => IO.pure(1)
  case _ =>
    for
      a <- IO.defer(fib(i - 1))
      b <- fib(i - 2)
    yield a + b
```

1

version #1 (**naïve**)
- not **tail-recursive** (not **stack-safe**)
- **exponential** time complexity
- **linear** stack frame depth

2

version #2 (**tupling**-based)
- not **tail-recursive** (not **stack-safe**)
- **linear** time complexity
- **linear** stack frame depth

3

version #3 (**accumulator**-based)
- **tail-recursive** (**stack-safe**)
- **linear** time complexity

4

version #4 (**left fold**-based)
- **non-recursive** (**stack-safe**)
- **linear** time complexity

5

version #4 (**stack-safe naïve** - **Eval**-based)
- not **tail-recursive** but **stack-safe**
- **exponential** time complexity

6

version #5 (**stack-safe naïve** - **IO**-based)
- not **tail-recursive** but **stack-safe**
- **exponential** time complexity

See you in part 2, in which we generate **potentially infinite streams** of **Fibonacci numbers**.

@philip_schwarz