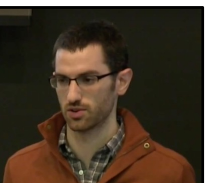
Fusing Transformations of Strict Scala Collections with Views

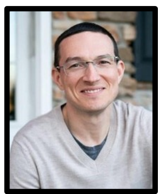
learn about it through the work of…

Harold Abelson

Gerald Jay Sussman

Runar Bjarnason


Paul Chiusano


Michael Pilquist

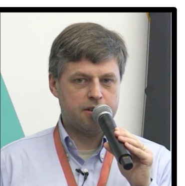
Li Haoyi

Martin Odersky

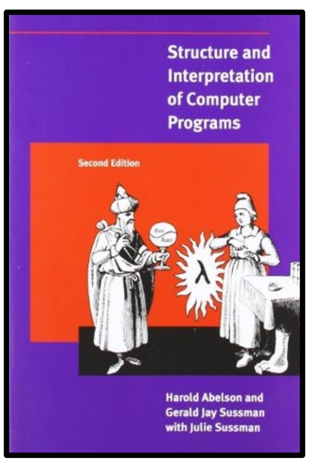
Lex Spoon

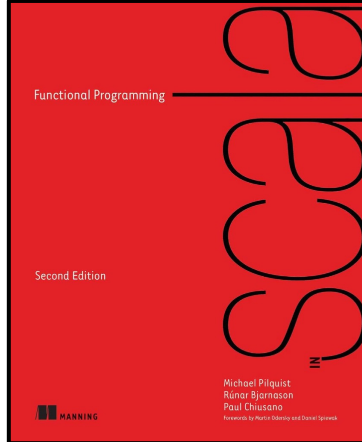
Frank Sommers

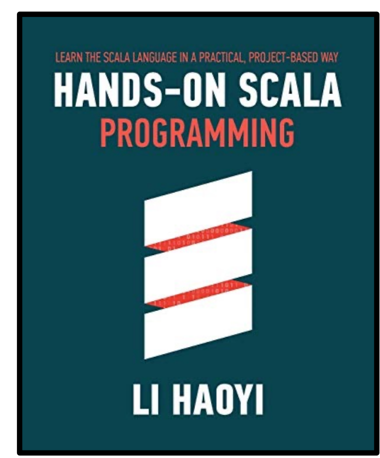
Bill Venners

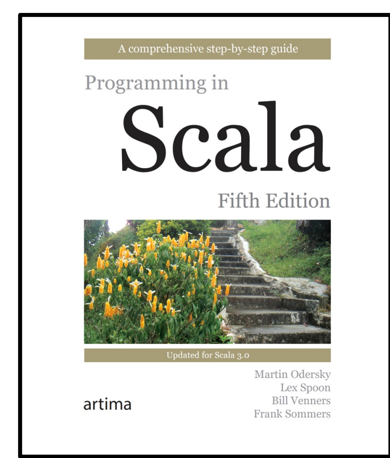

Sergei Winitzki

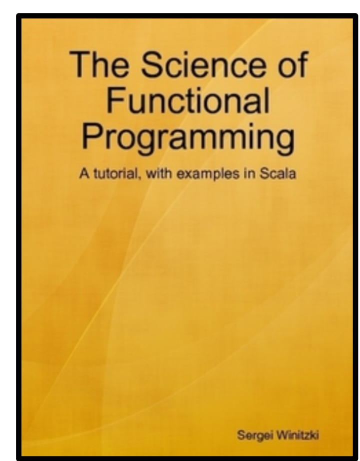
Alvin Alexander

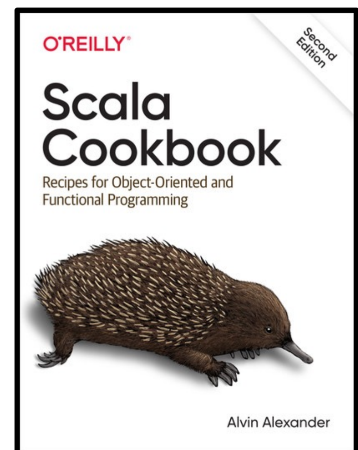












slides by @philip_schwarz http://fpilluminated.com/

This deck was inspired by the following tweet.

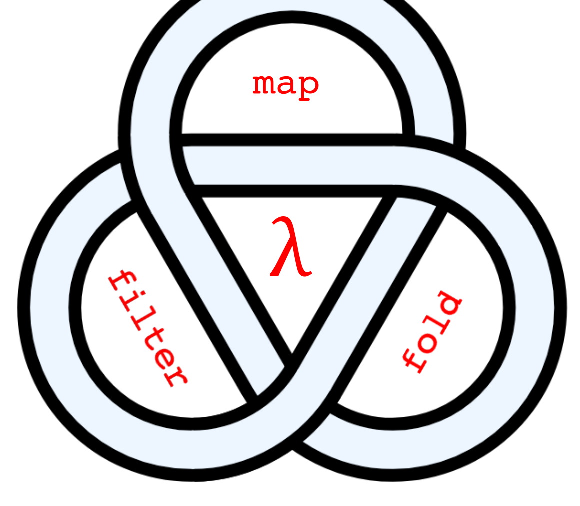
With excerpts from some great books, this deck aims to provide a good introduction to how the **transformations** of **eager collections**, e.g. **map** and **filter**, can be **fused** using **views**.

There is a lot of overlap between the subject of **views**, and that of **streams**. For example, they both address the problem of **fusing transformations**, and they both do it using **laziness**.

Because of that, I think it can be useful, before diving into the subject of **views**, to first go through a brief refresher on (or introduction to) **streams**.

If you prefer to go straight to the main subject of the deck then you can just jump to slide 13.

Among the many **transformer methods** of collections (methods that create new collections), **map** and **filter** stand out as members of the following triad that is the bread, butter and jam of functional programming.



```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate *
              1
              (map square
                   (filter odd? sequence))))
```

```
(defn product-of-squares-of-odd-elements [sequence]
  (accumulate *
              1
              (map square
                   (filter odd? sequence))))
```

```
def product_of_squares_of_odd_elements(sequence: List[Int]): Int =
  sequence.filter(isOdd)
          .map(square)
          .foldRight(1)(_*_)
```

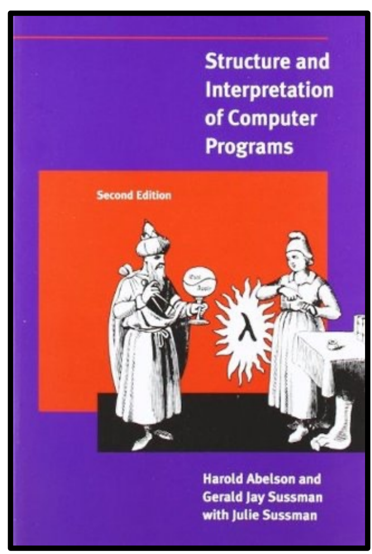
```
product_of_squares_of_odd_elements :: [Int] -> Int
product_of_squares_of_odd_elements sequence =
  foldr (*)
        1
        (map square
             (filter is_odd
                     sequence))
```

```
product_of_squares_of_odd_elements : [Nat] -> Nat
product_of_squares_of_odd_elements sequence =
  foldRight (*)
            1
            (map square
                 (filter is_odd
                         sequence))
```

I first came across the **map**, **filter** and **fold** triad many years ago, at university, while reading **Structure and Interpretation of Computer Programs** (**SICP**).

The following four slides consist of **SICP** excerpts introducing the following two ideas:

1) **Transformations** of **strict**/**eager** **lists** are severely **inefficient**, because they require **creation** and **copying** of data structures (that may be huge) at every step of a process (sequence of **transformations**).
2) **Streams** are a clever idea that exploits **delayed evaluation** (**non-strictness**/**laziness**) to permit the use of **transformations** without incurring some of the costs of manipulating **strict**/**eager lists**.



3.5 Streams

...

From an abstract point of view, a stream is simply a sequence.

However, we will find that the straightforward implementation of streams as lists (as in section 2.2.1) doesn't fully reveal the power of stream processing.

2.2.1 Representing Sequences

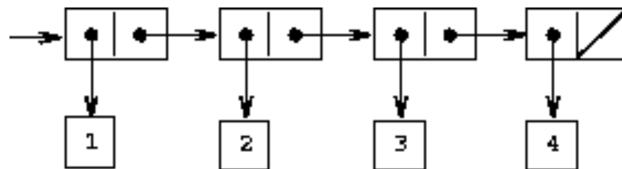


Figure 2.4: The sequence 1,2,3,4 represented as a chain of pairs.

One of the useful structures we can build with pairs is a *sequence* -- an ordered collection of data objects. There are, of course, many ways to represent sequences in terms of pairs. One particularly straightforward representation is illustrated in figure 2.4, where the sequence 1, 2, 3, 4 is represented as a chain of pairs. The car of each pair is the corresponding item in the chain, and the cdr of the pair is the next pair in the chain. The cdr of the final pair signals the end of the sequence by pointing to a distinguished value that is not a pair, represented in box-and-pointer diagrams as a diagonal line and in programs as the value of the variable nil. The entire sequence is constructed by nested cons operations:

```
(cons 1
      (cons 2
            (cons 3
                  (cons 4 nil))))
```

As an alternative, we introduce the technique of *delayed evaluation*, which enables us to represent very large (even infinite) sequences as streams. Stream processing lets us model systems that have state without ever using assignment or mutable data.

Structure and Interpretation of Computer Programs

3.5.1 Streams Are Delayed Lists

As we saw in section 2.2.3, sequences can serve as standard interfaces for combining program modules. We formulated powerful abstractions for manipulating sequences, such as map, filter, and accumulate, that capture a wide variety of operations in a manner that is both succinct and elegant.

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))
```

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

```
(define (filter predicate sequence)
        (cond ((null? sequence) nil)
              ((predicate (car sequence))
               (cons (car sequence)
                     (filter predicate (cdr sequence))))
              (else (filter predicate (cdr sequence)))))
```

$$
\begin{aligned}
&filter :: (\alpha \to Bool) \to [\alpha] \to [\alpha] \\
&filter\ p\ [\,] = [\,] \\
&filter\ p\ (x : xs) = \mathbf{if}\ p\ x \\
&\qquad \mathbf{then}\ x : filter\ p\ xs \\
&\qquad \mathbf{else}\ filter\ p\ xs
\end{aligned}
$$

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
```

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

Unfortunately, if we represent sequences as lists, this elegance is bought at the price of severe inefficiency with respect to both the time and space required by our computations. When we represent manipulations on sequences as transformations of lists, our programs must construct and copy data structures (which may be huge) at every step of a process.

Structure and Interpretation of Computer Programs

To see why this is true, let us compare two programs for computing the sum of all the prime numbers in an interval. The first program is written in standard iterative style:[53]

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

The second program performs the same computation using the sequence operations of section 2.2.3:

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime? (enumerate-interval a b))))
```

In carrying out the computation, the first program needs to store only the sum being accumulated. In contrast, the filter in the second program cannot do any testing until enumerate-interval has constructed a complete list of the numbers in the interval.

The filter generates another list, which in turn is passed to accumulate before being collapsed to form a sum.

Such large intermediate storage is not needed by the first program, which we can think of as enumerating the interval incrementally, adding each prime to the sum as it is generated.

[53] Assume that we have a predicate **prime?** (e.g., as in section 1.2.6) that tests for **primality**.

Structure and Interpretation of Computer Programs

The inefficiency in using lists becomes painfully apparent if we use the sequence paradigm to compute the second prime in the interval from 10,000 to 1,000,000 by evaluating the expression

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

This expression does find the second prime, but the computational overhead is outrageous. We construct a list of almost a million integers, filter this list by testing each element for primality, and then ignore almost all of the result.

In a more traditional programming style, we would interleave the enumeration and the filtering, and stop when we reached the second prime.

Streams are a clever idea that allows one to use sequence manipulations without incurring the costs of manipulating sequences as lists.

With streams we can achieve the best of both worlds: We can formulate programs elegantly as sequence manipulations, while attaining the efficiency of incremental computation.

The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream.

If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself to produce the required part, thus preserving the illusion that the entire stream exists.

In other words, although we will write programs as if we were processing complete sequences, we design our stream implementation to automatically and transparently interleave the construction of the stream with its use.

Structure and Interpretation of Computer Programs

Chapter 5 of **Functional Programming in Scala** explains how **lazy lists** (**streams**) can be used to **fuse** together sequences of **transformations**.

See the next slide for how the book introduces the problem that **streams** are meant to solve.

Chapter 5 - Strictness and laziness

In chapter 3 we talked about purely functional data structures, using singly linked lists as an example. We covered a number of bulk operations on lists—map, filter, foldLeft, foldRight, zipWith, and so on. We noted that each of these operations makes its own pass over the input and constructs a fresh list for the output.

Imagine if you had a deck of cards and you were asked to remove the odd-numbered cards and then flip over all the queens. Ideally, you'd make a single pass through the deck, looking for queens and odd-numbered cards at the same time. This is more efficient than removing the odd cards and then looking for queens in the remainder. And yet the latter is what Scala is doing in the following code:

```
scala> List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).map(_ * 3)
List(36,42)
```

In this expression, `map(_ + 10)` will produce an intermediate list that then gets passed to `filter(_ % 2 == 0)`, which in turn constructs a list that gets passed to `map(_ * 3)`, which then produces the final list. In other words, each transformation will produce a temporary list that only ever gets used as input to the next transformation and is then immediately discarded.

…

This view makes it clear how the calls to `map` and `filter` each perform their own traversal of the input and allocate lists for the output. Wouldn't it be nice if we could somehow fuse sequences of transformations like this into a single pass and avoid creating temporary data structures?

We could rewrite the code into a `while` loop by hand, but ideally we'd like to have this done automatically while retaining the same highlevel compositional style. We want to compose our programs using higher-order functions like `map` and `filter` instead of writing monolithic loops.

It turns out that we can accomplish this kind of automatic loop fusion through the use of non-strictness (or, less formally, laziness). In this chapter, we'll explain what exactly this means, and we'll work through the implementation of a lazy list type that fuses sequences of transformations. Although building a "better" list is the motivation for this chapter, we'll see that non-strictness is a fundamental technique for improving on the efficiency and modularity of functional programs in general.

…

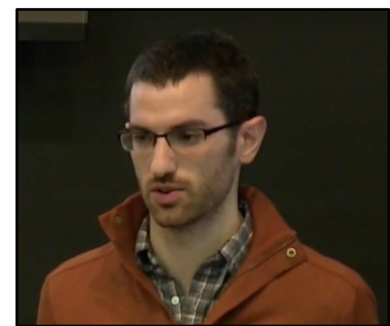
5.2 An extended example: lazy lists

Let's now return to the problem posed at the beginning of this chapter. We'll explore how laziness can be used to improve the efficiency and modularity of functional programs using lazy lists, or streams, as an example. We'll see how chains of transformations on streams are fused into a single pass through the use of laziness. Here's a simple `Stream` definition…

…

Runar Bjarnason

Functional Programming In Scala



Paul Chiusano

@pchiusano

Michael Pilquist

@philip_schwarz

With that refresher on (or introduction to) **streams** out of the way, let's get back to the actual subject of this deck: **views**.

In my view (pun not intended) a great book to get started with is **Li Haoyi**'s **Hands-On Scala Programming**. I find its approach to introducing **views** unique, and I love how he includes diagrams to illustrate the concept.

4.1.3 Transforms

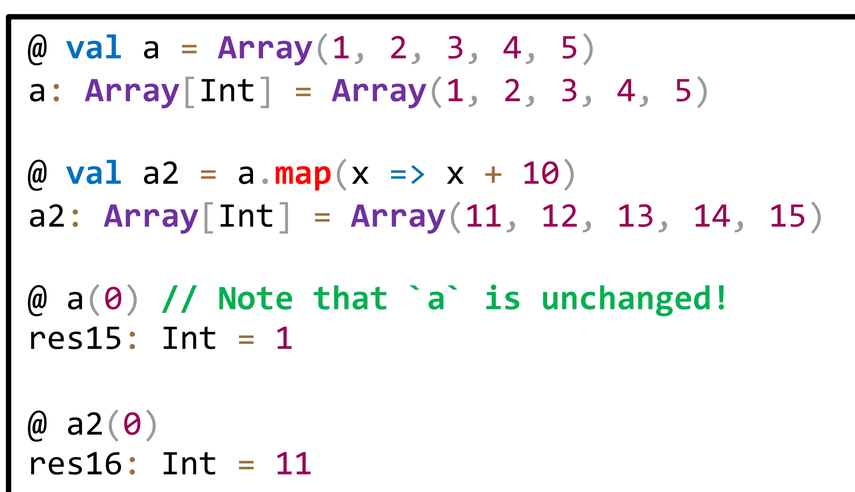
```
@ Array(1, 2, 3, 4, 5).map(i => i * 2) // Multiply every element by 2
res7: Array[Int] = Array(2, 4, 6, 8, 10)

@ Array(1, 2, 3, 4, 5).filter(i => i % 2 == 1) // Keep only elements not divisible by 2
res8: Array[Int] = Array(1, 3, 5)

@ Array(1, 2, 3, 4, 5).take(2) // Keep first two elements
res9: Array[Int] = Array(1, 2)

@ Array(1, 2, 3, 4, 5).drop(2) // Discard first two elements
res10: Array[Int] = Array(3, 4, 5)

@ Array(1, 2, 3, 4, 5).slice(1, 4) // Keep elements from index 1-4
res11: Array[Int] = Array(2, 3, 4)

@ Array(1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8).distinct // Removes all duplicates
res12: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8)
```

LEARN THE SCALA LANGUAGE IN A PRACTICAL, PROJECT-BASED WAY

HANDS-ON SCALA PROGRAMMING

LI HAOYI

Li Haoyi
@lihaoyi



```
@ val a = Array(1, 2, 3, 4, 5)
a: Array[Int] = Array(1, 2, 3, 4, 5)

@ val a2 = a.map(x => x + 10)
a2: Array[Int] = Array(11, 12, 13, 14, 15)

@ a(0) // Note that `a` is unchanged!
res15: Int = 1

@ a2(0)
res16: Int = 11
```

Transforms take an existing collection and create a new collection modified in some way. Note that these transformations create copies of the collection, and leave the original unchanged. That means if you are still using the original array, its contents will not be modified by the **transform**.

The copying involved in these collection transformations does have some overhead, but in most cases that should not cause issues. If a piece of code does turn out to be a bottleneck that is slowing down your program, you can always convert your .map/.filter/etc. transformation code into mutating operations over raw Arrays or In-Place Operations (4.3.4) over Mutable Collections (4.3) to optimize for performance.

4.1.6 Combining Operations

It is common to chain more than one operation together to achieve what you want. For example, here is a function that computes the standard deviation of an array of numbers:

```
@ def stdDev(a: Array[Double]): Double = {
  val mean = a.foldLeft(0.0)(_ + _) / a.length
  val squareErrors = a.map(_ - mean).map(x => x * x)
  math.sqrt(squareErrors.foldLeft(0.0)(_ + _) / a.length)
}
```

Scala collections provide a convenient helper method .sum that is equivalent to .foldLeft(0.0)(_ + _), so the above code can be simplified to…

As another example, here is a function that…

Chaining collection transformations in this manner will always have some overhead, but for most use cases the overhead is worth the convenience and simplicity that these transforms give you. If collection transforms do become a bottleneck, you can optimize the code using Views (4.1.8), In-Place Operations (4.3.4), or finally by looping over the raw Arrays yourself.

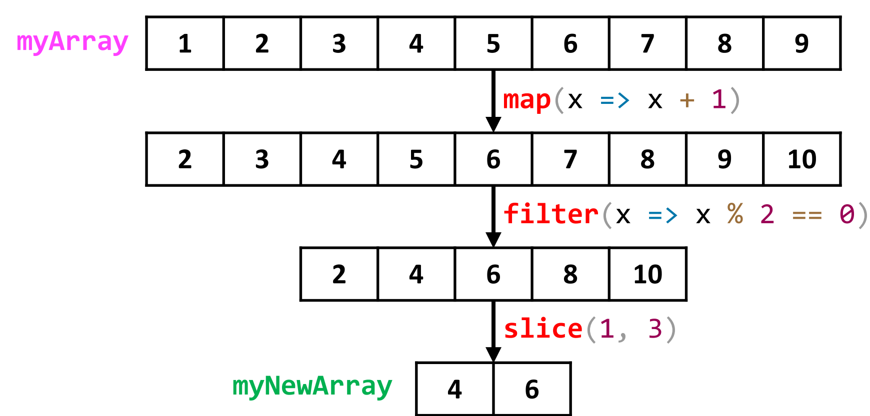
4.1.8 Views

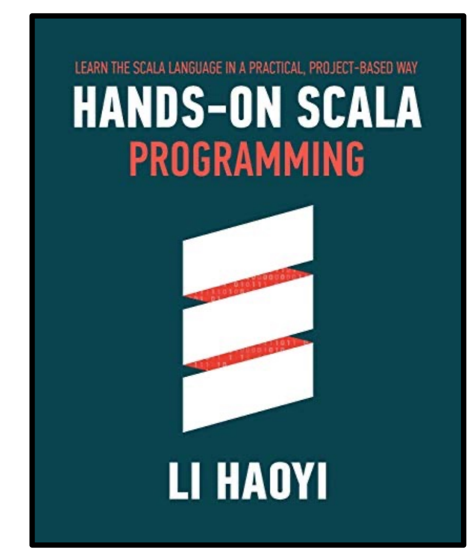
When you chain multiple transformations on a collection, we are creating many intermediate collections that are immediately thrown away. For example, in the following snippet:

```
@ val myArray = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)

@ val myNewArray = myArray.map(x => x + 1).filter(x => x % 2 == 0).slice(1, 3)
myNewArray: Array[Int] = Array(4, 6)
```

The chain of .map .filter .slice operations ends up traversing the collection three times, creating three new collections, but only the last collection ends up being stored in myNewArray and the others are discarded.

Li Haoyi
@lihaoyi



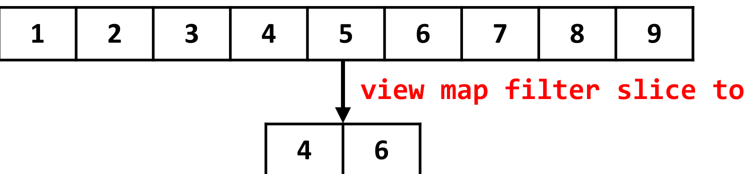


This creation and traversal of intermediate collections is wasteful. In cases where you have long chains of collection transformations that are becoming a performance bottleneck, you can use the .view method together with .to to "fuse" the operations together:

```
@ val myNewArray = myArray.view.map(_ + 1).filter(_ % 2 == 0).slice(1, 3).to(Array)
myNewArray: Array[Int] = Array(4, 6)
```

Li Haoyi
@lihaoyi

Using .view before the map/filter/slice transformation operations defers the actual traversal and creation of a new collection until later, when we call .to to convert it back into a concrete collection type:

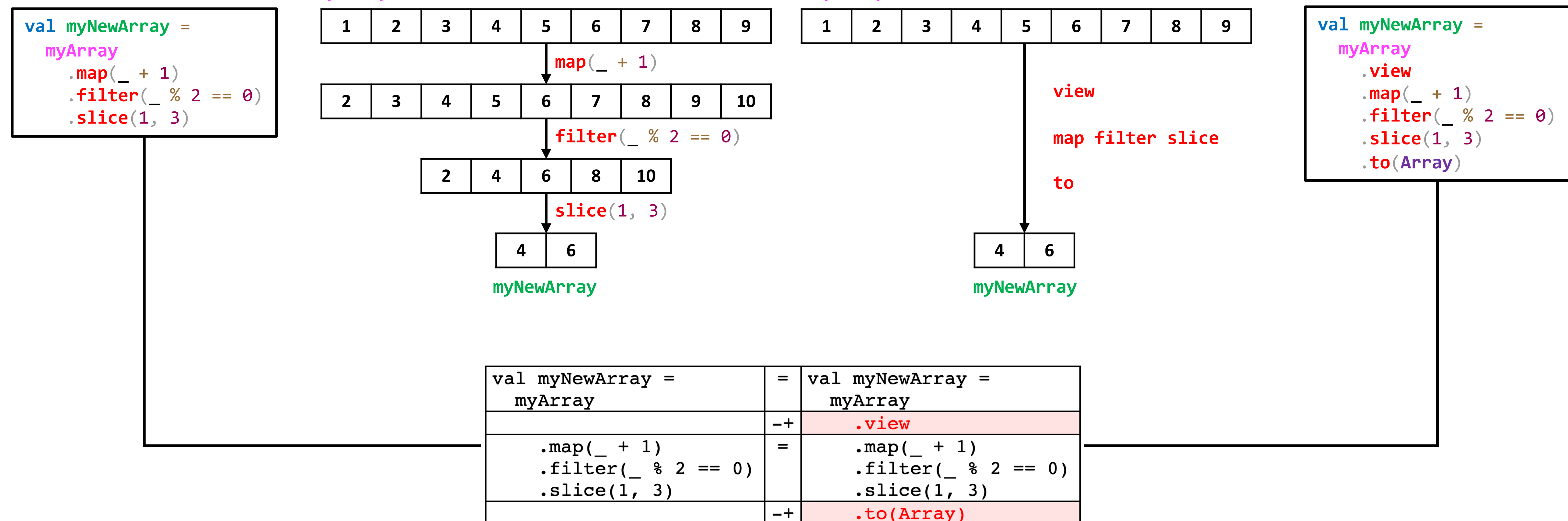


This allows us to perform this chain of map/filter/slice transformations with only a single traversal, and only creating a single output collection. This reduces the amount of unnecessary processing and memory allocations.

The next slide is just a quick visual recap.

- 3 traversals
- 3 collections created (2 intermediate ones)

- 1 traversal
- 1 collection created (no intermediate ones)
- call to **view** fuses **map filter** and **slice** together
- traversal is deferred until conversion to a concrete collection using **.to**



Next, let's turn to 'The' **Scala** book.

24.13 Views

Collections have quite a few methods that construct new collections. Some examples are map, filter, and ++.

We call such methods transformers because they take at least one collection as their receiver object and produce another collection in their result.

Transformers can be implemented in two principal ways: strict and nonstrict (or lazy).

A strict transformer constructs a new collection with all of its elements.

A non-strict, or lazy, transformer constructs only a proxy for the result collection, and its elements are constructed on demand.

As an example of a non-strict transformer, consider the following implementation of a lazy map operation:

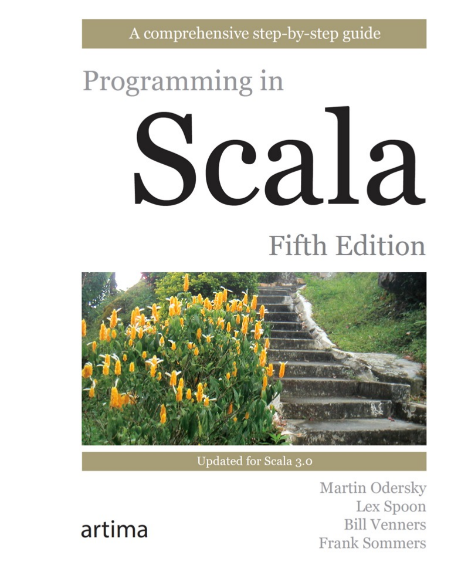
```
def lazyMap[T, U](col: Iterable[T], f: T => U) =
  new Iterable[U]:
    def iterator = col.iterator.map(f)
```

Note that `lazyMap` constructs a new `Iterable` without stepping through all elements of the given collection coll. The given function f is instead applied to the elements of the new collection's iterator as they are demanded.

Scala collections are by default strict in all their transformers, except for `LazyList`, which implements all its transformer methods lazily.

However, there is a systematic way to turn every collection into a lazy one and vice versa, which is based on collection views. A view is a special kind of collection that represents some base collection, but implements all of its transformers lazily.

To go from a collection to its view, you can use the view method on the collection. If xs is some collection, then xs.view is the same collection, but with all transformers implemented lazily. To get back from a view to a strict collection, you can use the to conversion operation with a strict collection factory as parameter.



Martin Odersky
@odersky

As an example, say you have a vector of Ints over which you want to map two functions in succession:

```
val v = Vector((1 to 10)*)
// Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
v.map(_ + 1).map(_ * 2)
// Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

In the last statement, the expression v.map(_ + 1) constructs a new vector that is then transformed into a third vector by the second call to map(_ * 2).

In many situations, constructing the intermediate result from the first call to map is a bit wasteful. In the pseudo example, it would be faster to do a single map with the composition of the two functions (_ + 1) and (_ * 2).

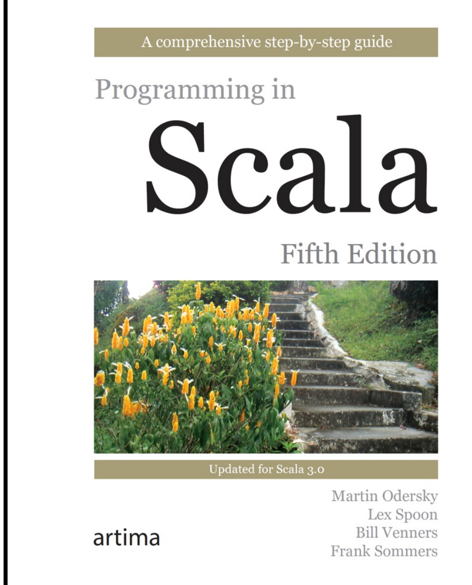
If you have the two functions available in the same place you can do this by hand. But quite often, successive transformations of a data structure are done in different program modules. Fusing those transformations would then undermine modularity. A more general way to avoid the intermediate results is by turning the vector first into a view, applying all transformations to the view, and finally forcing the view to a vector:

```
(v.view.map(_ + 1).map(_ * 2)).toVector
// Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

We'll do this sequence of operations again, one by one:

```
scala> val vv = v.view
val vv: scala.collection.IndexedSeqView[Int] = IndexedSeqView(<not computed>)
```

The application v.view gives you an IndexedSeqView, a lazily evaluated IndexedSeq. As with LazyList, toString on views does not force the view elements. That's why the vv's elements are displayed as not computed.



Bill Venners
@bvenners

Applying the first map to the view gives you:

```
scala> vv.map(_ + 1)
val res13: scala.collection.IndexedSeqView[Int] = IndexedSeqView(<not computed>)
```

The result of the map is another `IndexedSeqView[Int]` value. This is in essence a wrapper that records the fact that a map with function (_ + 1) needs to be applied on the vector v. It does not apply that map until the view is forced, however.

We'll now apply the second map to the last result.

```
scala> res13.map(_ * 2)
val res14: scala.collection.IndexedSeqView[Int] = IndexedSeqView(<not computed>)
```

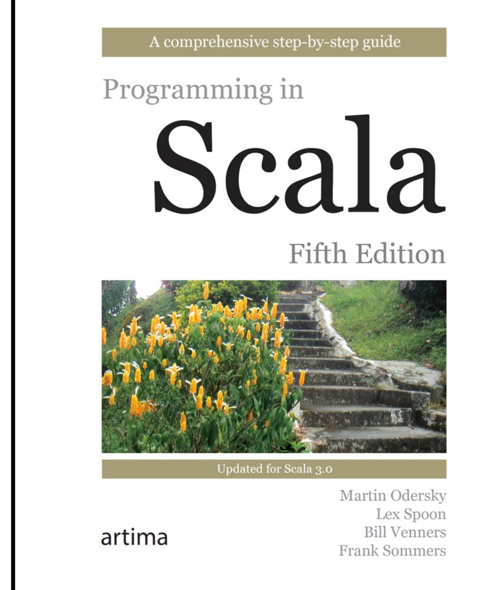
Finally, forcing the last result gives:

```
scala> res14.toVector
val res15: Seq[Int] = Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Both stored functions, (_ + 1) and (_ * 2), get applied as part of the execution of the to operation and a new vector is constructed. That way, no intermediate data structure is needed.

Transformation operations applied to views don't build a new data structure. Instead, they return an iterable whose iterator is the result of applying the transformation operation to the underlying collection's iterator.

The main reason for using views is performance. You have seen that by switching a collection to a view the construction of intermediate results can be avoided. These savings can be quite important.



Lex Spoon

As another example, consider the problem of finding the first palindrome in a list of words. A palindrome is a word that reads backwards the same as forwards. Here are the necessary definitions:

```
def isPalindrome(x: String) = x == x.reverse

def findPalindrome(s: Iterable[String]) = s.find(isPalindrome)
```

Now, assume you have a very long sequence, `words`, and you want to find a palindrome in the first million words of that sequence. Can you re-use the definition of `findPalindrome`? Of course, you could write:

```
findPalindrome(words.take(1000000))
```

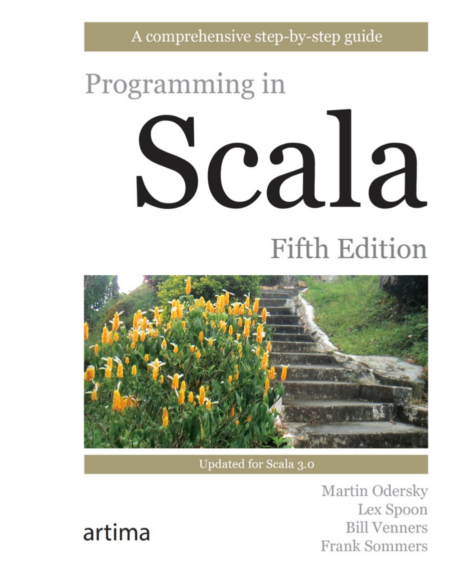
This nicely separates the two aspects of taking the first million words of a sequence and finding a palindrome in it. But the downside is that it always constructs an intermediary sequence consisting of one million words, even if the first word of that sequence is already a palindrome. So potentially, 999,999 words are copied into the intermediary result without being inspected at all afterwards. Many programmers would give up here and write their own specialized version of finding palindromes in some given prefix of an argument sequence. But with views, you don't have to.

Simply write:

```
findPalindrome(words.view.take(1000000))
```

This has the same nice separation of concerns, but instead of a sequence of a million elements it will only construct a single lightweight view object. This way, you do not need to choose between performance and modularity.

After having seen all these nifty uses of views you might wonder why have strict collections at all? One reason is that performance comparisons do not always favor lazy over strict collections. For smaller collection sizes the added overhead of forming and applying closures in views is often greater than the gain from avoiding the intermediary data structures. A possibly more important reason is that evaluation in views can be very confusing if the delayed operations have side effects.



Frank Sommers

In the next excerpt, **Sergei Winitzki**'s coverage of views, in the **Science of Functional Programming**, is short and sweet, but also nice and practical, with an emphasis on memory usage.

The code `(1L to 1_000_000_000L).sum` works because `(1 to n)` produces a sequence whose elements are computed whenever needed but do not remain in memory. This can be seen as a sequence with the "on-call" availability of elements. Sequences of this sort are called iterators:

```
scala> 1 to 5
res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> 1 until 5
res1: scala.collection.immutable.Range = Range(1, 2, 3, 4)
```

The types `Range` and `Range.Inclusive` are defined in the Scala standard library and are iterators. They behave as collections and support the usual methods (map, filter, etc.), but they do not store previously computed values in memory.

The view method

Eager collections such as List or Array can be converted to iterators by using the view method. This is necessary when intermediate collections consume too much memory when fully evaluated.

For example, consider the computation of Example 2.1.5.7 where we used flatMap to replace each element of an initial sequence by three new numbers before computing max of the resulting collection. If instead of three new numbers we wanted to compute three million new numbers each time, the intermediate collection created by flatMap would require too much memory, and the computation would crash:

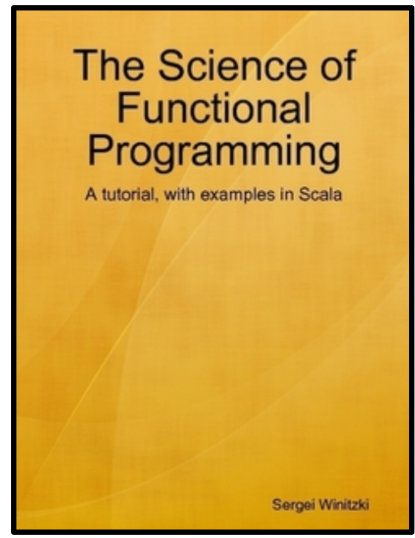
```
scala> (1 to 10).flatMap(x => 1 to 3_000_000).max
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

NOTES (from 'Programming in Scala'): In Scala versions before 2.8, the Range type was lazy, so it behaved in effect like a view. Since 2.8, all collections except lazy lists and views are strict. The only way to go from a strict to a lazy collection is via the view method. The only way to go back is via to.

Even though the range `(1 to 10)` is an iterator, a subsequent flatMap operation creates an intermediate collection that is too large for our computer's memory.

We can use view to avoid this:

```
scala> (1 to 10).view.flatMap(x => 1 to 3_000_000).max
res0: Int = 3_000_000
…
```



Sergei Winitzki

in sergei-winitzki-11a6431

And last, but not by no means least, let's look at **Alvin Alexander**'s **Scala Cookbook** recipe for **Creating a Lazy View on a Collection**.

11.4 Creating a Lazy View on a Collection

Problem

You're working with a large collection and want to create a lazy version of it so it will only compute and return results as they are needed.

Solution

Create a *view* on the collection by calling its view method. That creates a new collection whose *transformer methods* are implemented in a nonstrict, or lazy, manner. For example, given a large list:

```
val xs = List.range(0, 3_000_000) // a list from 0 to 2,999,999
```

imagine that you want to call several transformation methods on it, such as map and filter. This is a contrived example, but it demonstrates a problem:

```
val ys = xs.map(_ + 1).map(_ * 10).filter(_ > 1_000).filter(_ < 10_000)
```

If you attempt to run that example in the REPL, you'll probably see this fatal "out of memory" error:

```
scala> val ys = xs.map(_ + 1) java.lang.OutOfMemoryError: GC overhead limit exceeded
```

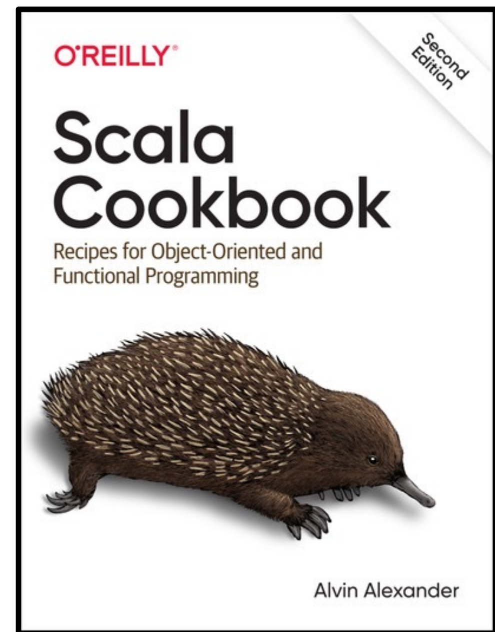
Conversely, this example returns almost immediately and doesn't throw an error because all it does is create a view and then four lazy transformer methods:

```
val ys = xs.view.map(_ + 1).map(_ * 10).filter(_ > 1_000).filter(_ < 10_000)
// result: ys:   scala.collection.View[Int] = View(<not computed>)
```

Now you can work with ys without running out of memory:

```
scala> ys.take(3).foreach(println)
1010
1020
1030
```

Calling view on a collection makes the resulting collection lazy. Now when transformer methods are called on the view, the elements will only be calculated as they are accessed, and not "eagerly," as they normally would be with a strict collection.



Alvin Alexander
@alvinalexander

Discussion

…

The use case for views

The main use case for using a view is performance, in terms of speed, memory, or both.

Regarding performance, the example in the Solution first demonstrates (a) a strict approach that runs out of memory, and then (b) a lazy approach that lets you work with the same dataset.

The problem with the first solution is that it attempts to create new, intermediate collections each time a transformer method is called:

```
val b = a.map(_ + 1)          // 1st copy of the data
         .map(_ * 10)         // 2nd copy of the data
         .filter(_ > 1_000)   // 3rd copy of the data
         .filter(_ < 10_000) // 4th copy of the data
```

If the initial collection a has one billion elements, the first map call creates a new intermediate collection with another billion elements.

The second map call creates another collection, so now we're attempting to hold three billion elements in memory, and so on.

To drive that point home, that approach is the same as if you had written this:

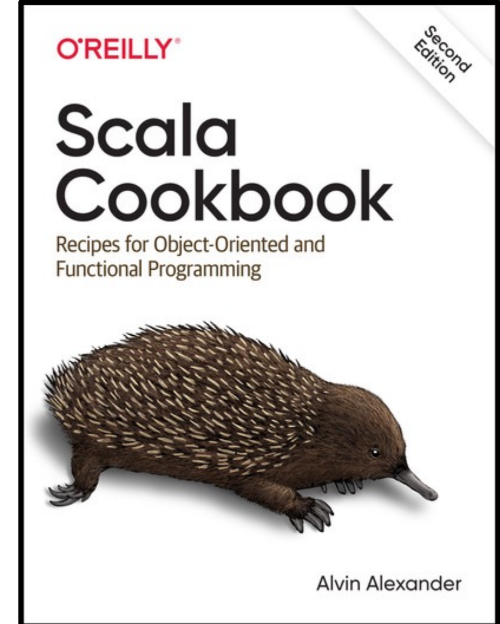
```
val a = List.range(0, 1_000_000_000) // 1B elements in RAM
val b = a.map(_ + 1)                 // 1st copy of the data (2B elements in RAM)
val c = b.map(_ * 10)                // 2nd copy of the data (3B elements in RAM)
val d = c.filter(_ > 1_000)          // 3rd copy of the data (~4B total)
val e = d.filter(_ < 10_000)         // 4th copy of the data (~4B total)
```

Conversely, when you immediately create a view on the collection, everything after that essentially just creates an iterator:

```
val ys = a.view.map ... // this DOES NOT create another one billion elements
```

As usual with anything related to performance, be sure to test using a view versus not using a view in your application to find what works best. Another performance-related reason to understand **views** is that it's become very common to work with large datasets in a streaming manner, and **views** work very similar to streams.

…



Alvin Alexander
@alvinalexander

That's all. I hope you found it useful.

@philip_schwarz