INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Inferring an Iterator from a bulk traversing function

DIOGO RAFAEL ESTEVES POEIRA

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Junho, 2018

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Inferring an Iterator from a bulk traversing function

DIOGO RAFAEL ESTEVES POEIRA

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Junho, 2018

Contents

Listings

1 Introduction

Extending java.util.stream with new query operations requires implementation of two distinct iteration protocols:

- Traversing elements individually (i.e. tryAdvance())
- Traversing elements in bulk (i.e. forEachRemaining())

This incurs in undesired verbosity. In this project we propose a solution - Jayield - which infers the individual element traversal definition from a given bulk traversal method, and thus allows the implementation of new query operations with a single method (e.g. lambda expression).

1.1 Motivation

Java 8 brings with it the java.util.stream API (henceforth named just Stream) which allows us to perform queries fluently on a sequences of data, by providing methods that accept behaviour as parameters, allowing the programmers to express what their intentions are when calling the method.

Let us consider the following two classes.

```java
public class Fruit {
   String name;
   String color;

   public Fruit(String name, String color) {
      this.name = name;
      this.color = color;
   }
}
```

Listing 1.1: Fruit

```java
import static java.util.Arrays.asList;

public class Basket {

   Fruit orange = new Fruit("Orange", "orange");
   Fruit lemon = new Fruit("Lemon", "yellow");
   ...
   Fruit mango = new Fruit("Mango", "orange");

   public final List<Fruit> fruits = asList(orange, lemon, mango, ...);

}
```

Listing 1.2: Basket

Listing 1.1 specifies the definition of a Fruit and listing 1.2 makes use of said definition and specifies the fruits inside a basket. Now, lets say we want to know the names of all the fruits inside the basket that have the color orange. In Java 7 and previous versions we would do something like this:

```java
public static List<String> getOrangeFruitList(Basket basket) {
   Iterator<Fruit> iterator = basket.fruits.iterator();
   List<String> result = new ArrayList<Fruit>();
   while (iterator.hasNext()) {
      Fruit current = iterator.next();
      if (current.color.equals("orange")) {
         result.add(current.name);
      }
   }
   return result;
}
```

Listing 1.3: Iterator approach

As we can see in, we would attain an iterator and as we iterate over the fruits on the basket we check their color, if it matches our desired color we add it's name to the list and when we are done, we return the new list. Now lets look at how we can do this in Java 8 making use of Streams.

```java
public static Stream<String> getOrangeFruitStream(Basket basket) {
   return basket.fruits.stream()
               .filter(current -> current.color.equals("orange"))
               .map(current -> current.name);

}
```

Listing 1.4: Stream approach

This approach lets us define our intentions, first we say we are only interested in fruits with the color orange, and that for each fruit that matches that description we are only interested in it's name. This approach lets us specify our intentions clearly which in turn makes the code easier to read and understand.

However, the end use API is not the only difference between these two approaches. The iterator approach is eager, meaning, it immediately goes through all the fruits in the basket to select the ones we are interested, taking note of the names as we go through them. The stream approach is lazy, as we said before, all we are doing is specifying our intention, no iteration is made on the sequence until it is consumed by a terminal operation, such as a forEach().

1.2 Traversing sequences

Going back to our initial example, the basket of fruits, we now have the names of all the orange fruits in our basket. We now find ourselves with an overwhelming amount of names, to solve this, we decide to just take every other name, reducing the number of names by half. However, the Stream API that Java 8 provides does not support this operation, so we have to implement it ourselves. To that end Java provides the Spliterator interface which we can implement or the Abstract-Spliterator class that we can extend.

In listing 1.5 we show an example of the implementation of a filterOdd operation (line 2) which in turn requires an auxiliary class implementing spliterator (i.e. OddFilter, lines 3 and 5). In this case we extend the AbstractSpliterator instead of implementing the Spliterator interface to reduce the number of methods we're required to implement.

```java
public class Example {
  static <T> Stream <T> filterOdd (Stream <T> source ) {
    return StreamSupport.stream(new OddFilter<>(source.spliterator()),
        false);
  }
  static class OddFilter <T> extends AbstractSpliterator <T> {
    final Consumer <T> doNothing = item -> {};
    final Spliterator <T> source ;
    boolean isOdd = false;
    public Odd (Spliterator <T> source) {
      super (odd(source.estimateSize()), source.characteristics());
      this.source = source;
    }
    @Override
    public boolean tryAdvance(Consumer <? super T> action ) {
      if (!source.tryAdvance(doNothing)) return false ;
        return source.tryAdvance(action);
    }
    @Override
    public void forEachRemaining(Consumer<? super T> action) {
      source.forEachRemaining(item -> {
        if(isOdd){
          action.accept(item);
        }
        isOdd = !isOdd;
      });
    }
    private static long odd( long l) {
      return l == Long.MAX_VALUE ? l : (l +1)/2;
    }
  }
  public static void main(String[] args) {
    filterOdd(getOrangeFruitStream(new Basket()))
          .forEach(System.out::println);
  }
}
```

Listing 1.5: Spliterator example

This approach is verbose, and only benefits from bulk traversing because we have overriden forEachRemaing(), as the Spliterator documentation states, the forEachRemaining() [8] method's default implementation should be overridden whenever possible as it just calls tryAdvance() until it returns false, in other words, it will traverse element by element instead of the whole sequence.

So what was the reasoning behind the Spliterator approach, why not just an Iterator? One of the reasons was to achieve parallelism in the iteration of the sequence. From a use case point of view, most of the time, we do not wish to parallelize work, most of our data isn't big enough to justify the parallelization.

Another reason was the implementation of short-circuiting operations, as we have two methods in Spliterator that we can override to traverse a sequence, tryAdvance() and forEachRemaining(), tryAdvance() is the stipulated to individually traverse a sequence and therefore the one that should be used when short-circuiting a traversal. However, there are other ways to stop the traversing of a sequence, namely we can simply raise an exception when bulk traversing and the traversal will stop. Granted, this is somewhat of a polemic approach, but there is no hard evidence that states that it isn't a valid one, in fact, Python uses this approach, and nowadays there are lightweight Exceptions that can be used to this effect.

Lastly, and from our point of view, the most valid reason, operations that combine two sequences. Operations such as Zip that require that an element from each sequence be taken to combine it into a single element, cannot be achieved by a bulk traversing function.

To overcome the shortcomings found in the Stream API we will implement a solution that allows the user to define a single way to traverse a sequence and we will generate it's counterpart. In this case the filterOdd() operation can be expressed in a lambda expression chained with the sequence of items as shown in listing 1.6.

```java
boolean[] isOdd = {false};
getOrangeFruitQuery(new Basket())
      .then(source -> yield -> source.traverse(item -> {
         if(isOdd[0]) {
            yield.ret(item);
         }
         isOdd[0] = !isOdd[0];
      }));
```

Listing 1.6: Jayield filterOdd

The programmer just needs to write one method to be able to traverse a sequence as the individually advance will be inferred from the previous definition.

1.3 Use Case

The use case that will be analysed, developed and tested will be the generation of an Advancer from either a Query or a Traverser.

The difficulty with this use case is the fact that a Traverser is used to bulk traverse an entire sequence, meaning it is not (or should not be) prepared to verify if it can advance at each element of the sequence. To do this we will copy the Traverser's code and instrument the required parts to generate a new Advancer. To test if the generated Advancer is working properly we will verify that each call to tryAdvance that succeeds, yields one element and only one element.

2 Alternatives

In order to further understand the context of this project we will be describing some alternatives to the Java 8 Stream API as well as discussing their advantages and disadvantages.

2.1 Guava

Guava [3] was for some years the library most programmers defaulted to when looking for Functional programming utilities in Java, until Java 8 arrived. To achieve this functionality the Guava library provides their own sequential type, FluentIterable, which provides a way to chain operations together like Stream does.

In listing 2.1 we show how the previous example, where we obtain the names of all the orange fruits, could be achieved using the Guava library and lambda functions.

```java
import com.google.common.collect.FluentIterable;

import static com.google.common.collect.FluentIterable.from;

public class GuavaExample {
   public FluentIterable<String> getOrangeFruits(Basket basket) {
      return from(basket.fruits)
            .filter(fruit -> fruit.color.equals("orange"))
            .transform(fruit -> fruit.name);
   }
}
```

Listing 2.1: Guava's approach

Just like the Stream approach, here we can specify our intentions without traversing the sequence. When it comes to extending FluentIterables's operations even further however, the programmer has to implement an Iterator, like with Streams one would have to implement a Spliterator. So if presented with the same outcome as before, an implementation of filterOdd would look something like what is shown in listing 2.2 which results in a quite verbose approach and not what we are aiming to achieve.

```java
import com.google.common.collect.FluentIterable;

import java.util.Iterator;
import java.util.NoSuchElementException;

import static com.google.common.collect.FluentIterable.from;

public class FilterOdd<T> implements Iterator<T> {
   public static <T> FluentIterable<T> filterOdd(FluentIterable<T>
       source) {
      return from(() -> new FilterOdd<>(source.iterator()));
   }

   private final Iterator<T> source;
   private T current;

   public FilterOdd(Iterator<T> source){
      this.source = source;
   }
```

```java

    @Override
    public boolean hasNext() {
        if(current == null && source.hasNext()) {
            source.next(); // ignore even
            if(source.hasNext()) {
                current = source.next(); // get odd
            }
        }
        return current != null;
    }

    @Override
    public T next() {
        if(hasNext()){
            T aux = current;
            current = null;
            return aux;
        } else {
            throw new NoSuchElementException();
        }
    }
}
```

Listing 2.2: Guava's approach

2.2 StreamEx

StreamEx [1] is the only library, of those discussed in this document, that provides function based extensibility, instead of requiring the implementation of a specific type. For instance, looking back at our examples, to implement a filterOdd operation we would only need to define what is needed for the tryAdvance method described in listing 1.5 which iterates over two items whenever the downstream request one item. Listing 2.3 shows an example.

```java
import one.util.streamex.StreamEx;

import java.util.Spliterator;
import java.util.function.Consumer;

import static one.util.streamex.StreamEx.of;
import static one.util.streamex.StreamEx.produce;

public class StreamExExample {

   public static <T> StreamEx<T> filterOdd(StreamEx<T> source) {
      final Consumer<T> doNothing = item -> {};
      final Spliterator<T> iterator = source.spliterator();
      return produce(action -> {
         if (!iterator.tryAdvance(doNothing))
            return false;
         return iterator.tryAdvance(action);
      });
   }

   public StreamEx<String> getOrangeFruits(Basket basket) {
      return of(basket.fruits)
            .filter(fruit -> fruit.color.equals("orange"))
            .map(fruit -> fruit.name);
   }
}
```

Listing 2.3: StreamEx's approach

This approach is less verbose than the one implemented by Guava and Java 8's Streams. However StreamEx's implementation still has part of the drawbacks presented by Java 8's Stream API:

- Extensibility of the API breaks the fluent API. Even though the extension of the API requires less code, it is not possible to chain together with an already existing StreamEx instance, which breaks the fluent API provided by this library.
- The flatMap implementation breaks any short-circuiting operations.

2.3 JOOL and Cyclops

JOOL [5] and Cyclops [6] are equivalent to Stream. Both these libraries provide their own sequential types, Seq for JOOL and ReactiveSeq for Cyclops. They both expand the catalogue of operations when comparing with Stream, but when it comes to extending either API the programmer still needs to implement a Spliterator just like the one presented in listing 1.5.

2.4 Vavr

VAVR [4] provides its own sequential type, Seq, and like Guava, to extend its API the programmer needs to implement an Iterator which makes impossible for fluent API extensions. This library provides the widest set of functional data structures and operations from the libraries we studied and is one of most popular libraries on Github, second only to Guava. It provides an implementation of flatMap that does not break a short-circuiting operation on an infinite sequence.

3 Solution

In this chapter we will present the Jayield solution, going through it's different components and understanding how they work and interact with each other.

3.1 Yield

Yield is the interface that represents the element generator in our solution. It's interface is equivalent to a Java Consumer, however, it has a few key differences. Listing 3.1 presents the definition of the Yield interface.

```java
public interface Yield<T> {
   static final TraversableFinishError finishTraversal = new
      TraversableFinishError();

   static public void bye() { throw finishTraversal; }

   public void ret(T item);
}
```

Listing 3.1: Yield

As we can see Yield has two methods:

- bye(), which is used to short circuit the traversal of a sequence.

- ret(T item), which is the method used to define what to do with each item that a sequence returns when traversing it.

Lets say we were traversing a sequence of Fruit names and we would like to write down their names, listing 3.2 shows what the Yield lambda to achieve this looks like.

```java
name -> System.out.println(name)
```

Listing 3.2: Yield example

3.2 Traverser

Traverser is an interface that is used to define ways of bulk travesing a sequence. To that end it provides the method traverse which receives a Yield function as argument to consume elements as they are traversed. Listing 3.3 shows the definition of the Traverser interface.

```java
public interface Traverser<T> extends Serializable {
   void traverse(Yield<? super T> yield);
}
```

Listing 3.3: Traverser

Looking at some examples on how to use this interface, if we have an upstream sequence of Fruits, listing 3.4 shows of how to transform such a sequence into a sequence of Fruit's names.

```java
public Traverser<String> mapToName(Traverser<Fruit> upstream) {
   return yield -> upstream.traverse(fruit -> yield.ret(fruit.name));
}
```

Listing 3.4: Traverser map to name example

We can also define new sequences with the Traverser interface, lets say we want to generate a sequence of Fruits, listing 3.5 shows an example of how to do it.

```java
public Traverser<Fruit> getFruitSequence() {
   return yield -> {
      yield.ret(Basket.orange);
      yield.ret(Basket.blueberry);
      ...
      yield.ret(Basket.lemon);
      yield.ret(Basket.mango);
      yield.ret(Basket.strawberry);
   };
}
```

Listing 3.5: Traverser as sequence generator

3.3 Query

Looking now at the Query class in Jayield, it represents a sequence of elements supporting sequential operations. This class implements all the utility methods with the same name, semantics and behaviour as those provided by Java's Stream. This class has an Traverser field that represents the current sequence. It also provides a way for the user to define new operations through the then method (listing 3.6).

```java
<R> Query<R> then(Function<Query<T>, Traverser<R>> next)
```

Listing 3.6: Query then method

This method receives a Function as parameter that defines the way to transform the upstream sequence (represented by a Query) into the new traversal function, a Traverser. This function is similar to the method presented in the previous section in listing 3.4, with that in listing 3.4 we receive a Traverser as a parameter and the Function parameter in next receives a Query. An example of how to use this method can be looked at back in Chapter 1's listing 1.6.

3.4 Advancer

The Advancer is the interface that represents our individual traverser and the main focus of the use case of we are implementing.

The implementation for the Advancer (i.e. the individual traverser) will only be generated if requested to the Advancer interface as it is not always necessary to traverse. To do so we can call one of the two from methods provided in Advancer as shown in the 3.7 listing.

```java
public interface Advancer<T> extends Serializable {

    static <U> Iterator<U> iterator(Query<U> source) {...}

    static <U> Advancer<U> from(Query<U> source) {...}

    static <U> Advancer<U> from(Traverser<U> source) {...}

    static <U> Iterator<U> iterator(Traverser<U> source) {...}

    default Iterator<T> iterator() {...}

    default <U> Advancer<U> then(Function<Advancer<T>,Advancer<U>> next)
        {...}

    boolean tryAdvance(Yield<T> yield);

}
```

Listing 3.7: Advancer

This interface provides a way for the programmer to individually traverse a sequence, either through the tryAdvance method or through the use of an iterator. For instance, if we wanted to print the name of the first Fruit in a sequence we could take the approach shown in listing 3.8 which will print the first fruit's name, and only the first, if it exists.

```java
void printFirstName(Query<Fruit> fruits) {
    Advancer.from(fruits)
            .tryAdvance(fruit -> System.out.println(fruit.name));
}
```

Listing 3.8: Advancer example

So, the question now is, how do we generate an Advancer from a Query or from a Traverser?

3.4.1 Approach

To achieve the expected result a pseudo-solution was thought of, assuming one of the following is true:

- A call to source.traverse(...) is made
- A call to yield.ret(...) is made

Assuming any of these are true, we would generate an Advancer doing the following:

- Instead of calling source.traverse(...) we would need to call source.tryAdvance(...)
- When calling yield.ret(...) we would register that an element has been found that meets the criteria.
- The call to source.tryAdvance(...) is made like so: while(noElementFound && source.tryAdvance(...))

So, lets say we have the following definition for a map Traverser:

```java
<T, R> Traverser<R> map(Query<T> source, Function<T, R> mapper) {
 return yield -> {
    source.traverse(e -> yield.ret(mapper.apply(e)));
 };
}
```

Listing 3.9: Map Traverser

Given the bulk traversal definition of map in listing 3.9, then the resulting map which implements the individual traversal is shown in listing 3.10

```java
<T, R> Advancer<R> map(Query<T> source, Function<T, R> mapper) {
 boolean[] noElementFound = new boolean[] {true};
 return yield -> {
   noElementFound[0] = true;
   while(noElementFound[0] && source.tryAdvance(e -> {
      yield.ret(mapper.apply(e)));
      noElementFound[0] = false;
    });
   return noElementFound[0] == false;
 };
}
```

Listing 3.10: Map Advancer

With that said, there are corner cases that are not so easily mapped as is the case of the flatMap operation which traverses over multiple sequences, these cases need a different approach.

Nevertheless, following the suggested approach we will need to generate code at runtime, to do this we will use ASM, a library for java bytecode manipulation at runtime. In order to read the bytecode via ASM we first need to give it (via the ClassReader) the actual bytecode of the Traverser. The problem is, if the Traverser was generated using a lambda function, that the lambda's class is generated at runtime via the LambdaMetaFactory class and therefore the normal method of getting the bytecode does not work, aside from that we also needed to get any fields the Traverser's class may have, in the case of the lambda, they would be captured arguments.

3.4.2 SerializedLambda

You may have noticed in listing 3.3 that the Traverser interface extends from Serializable, the fact is, we actually need the Traverser to extend Serializable in order to read its code. This solution was found in an answer to a question in Stackoverflow[2],which makes use of the SerializedLambda[7].

The reason for this is that, if the lambda we are trying to read implements an interface that extends Serializable, the LambdaMetaFactory class will generate a private method called "writeReplace" which provides an instance of the class SerializedLambda. This instance can be used to retrieve both the actual method that was generated for this lambda as well as get the captured arguments and being able to reach the lambda's method allow us to instrument it.

3.4.3 Visitors

Now having access to the lambda's code, it was time to instrument it in order to transform the Traverser into an Advancer. To do so, we started by using a custom ClassVisitor that copies the original declaring class of the lambda as well as making some changes along the way, such as:

- The name of the class would now be the name of the lambda's method.
- Each call to a method on the declared class would now be a call to the generated class, due to calls to private methods.

- The return type of the lambda's method would now be boolean instead of void, to be coherent with the Advancer Interface.
- The lambda's method would now call tryAdvance instead of traverse and return the result of tryAdvance as well.

To obtain an Advancer out of the generated class all we would need to do now would be:

- Retrieve the original lambda's captured arguments
- Retrieve the generated method from the new class
- Apply the approach defined above.

Which would result in something like this:

```java
final BoolBox elementFound = new BoolBox();
Method generatedTryAdvance = getTryAdvance(...);
Advancer generated = y -> {
  elementFound.reset();
  Yield<R> wrapper = wr -> {
    y.ret(wr);
    elementFound.set();
  };
  arguments[arguments.length - 1] = wrapper;
  while(elementFound.isFalse() && (Boolean) generatedTryAdvance.invoke(
      newClass, arguments));
  return elementFound.isTrue();
};
```

Listing 3.11: Generated Advancer

This solution makes use of a wrapper over the yield, received as parameter, to register when an element is found, this wrapper is then provided as the last argument in an argument array for the generated method, with the first parameters being those captured by the lambda.

3.4.4 State Machine

This approach however, as said before, has flaws and the above described solution does not work for every case. One of the first issues identified was the case functions that would generate more elements than the original sequence, that being either, a duplicate function or a generator of values. Lets look at listing 3.12 as an example of an traverser that duplicates values.

```java
public <T> Traverser<T> dup(Traverser<T> upstream) {
   return yield -> upstream.traverse(item -> {
      yield.ret(item);
      yield.ret(item);
   });
}
```

Listing 3.12: Duplicate Traverser function

For this Traverser, the solution described above would not work as it would not stop when the first element was found nor would it return to the same point when the next element was requested. To solve this issue two things were necessary, a state machine and external iteration.

The first approach to this issue was based on declaring the end of a state, and the beginning of the next, every time a yield.ret(...) was made. This would translate to something like what is shown in listing 3.13.

```java
yield -> {
   this.hasElement.reset();
   YieldWrapper wrapper = new YieldWrapper(yield, this.hasElement);
   Query source = this.source;
   Yield instrumentedYield = (item) -> {
      if(this.state == 0){
         this.state++;
         wrapper.ret(item);
         return;
      } else if (this.state == 1) {
         this.state++;
         wrapper.ret(item);
         return;
      } else if (this.state == 2) {
         this.state = 0;
         this.validValue = false;
         return;
      }
   };

   while(this.hasElement.isFalse() && (this.iterator.hasNext() || this.
       validValue)) {
      if (!this.validValue && iterator.hasNext()) {
         this.current = iterator.next();
```

```
            this.validValue = true;
         }
         instrumentedYield.ret(this.current);
      }

      return this.hasElement.isTrue() ? true : this.hasElement.isTrue();
}
```

Listing 3.13: Duplicate Advancer translation with if State Machine

As you can see there are a few key differences:

- the instrumented yield operation now isolates yield.ret() calls into different states
- the validValue field, which lets us know if we need to get a new value or if the old one still needs to be used in another state.

This solution had it's problems aswell, for instance, if a yield return was made inside an if condition it would have to take that into account which would lead to even more complicated translations.

At this moment we took a step back to re-evaluate and it was then that we started to look into Jon Skeet's C# in Depth[9], chapter 6. The approach described here, makes use of a switch case as well as labels, with the labels being the delimiters between the end of a state and the beginning of a new one. This approach requires some pre-processing of the Traverser code, to know how many states there will be in the switch as in Java to declare a switch we need to know in advance how many states there will be, this approach was also applied to value generating lambdas, as the same principles apply. Listing 3.14 shows an example of the translation of the dup operation.

```
yield -> {
   this.hasElement.reset();
   YieldWrapper wrapper = new YieldWrapper(yield, this.hasElement);
   Query source = this.source;
   Yield instrumentedYield = (item) -> {
      switch(this.state) {
         case 0:
            wrapper.ret(item);
            this.state = 1;
            return;
         case 1:
            wrapper.ret(item);
```

```
                this.state = 2;
                return;
            case 2:
                this.state = 0;
                this.validValue = false;
                return;
            default:
        }
    };

    while(this.hasElement.isFalse() && (this.iterator.hasNext() || this.validValue)) {
        if (!this.validValue && iterator.hasNext()) {
            this.current = iterator.next();
            this.validValue = true;
        }
        instrumentedYield.ret(this.current);
    }

    return this.hasElement.isTrue() ? true : this.hasElement.isTrue();
}
```

Listing 3.14: Duplicate Advancer translation with Switch based State Machine

Bibliography

[1] amaembo. Streamex. URL https://github.com/amaembo/streamex. (p. 9)

[2] bennyl. How to read lambda expression bytecode using asm, March 2017. URL https://stackoverflow.com/a/35223119. (p. 18)

[3] Kevin Bourrillion and Google Jared Levy. Guava, 2009. URL https://github.com/google/guava. (p. 7)

[4] Daniel Dietrich. Cyclops, 2014. URL https://github.com/vavr-io/vavr. (p. 11)

[5] Lukas Eder. jool, 2014. URL https://github.com/jOOQ/jOOL. (p. 11)

[6] John McClean. Cyclops, 2014. URL https://github.com/aol/cyclops-react/. (p. 11)

[7] Oracle. Serialized lambda, . URL https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/SerializedLambda.html. (p. 18)

[8] Oracle. Spliterator, . URL https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#forEachRemaining-java.util.function.Consumer-. (p. 4)

[9] Jon Skeet. C# in depth, 2013. URL http://csharpindepth.com/Articles/Chapter6/IteratorBlockImplementation.aspx. (p. 21)