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() [6] 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 [2] 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.

```
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.

```
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;
   }
```

```

    @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 [4] and Cyclops [5] 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 [3] 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.

Bibliography

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

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

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

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

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

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