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

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.

```
1 public class Fruit {
2    String name;
3    String color;
4
5    public Fruit(String name, String color) {
6       this.name = name;
7       this.color = color;
8    }
9 }
```

Listing 1.1: Fruit

```
1 public class Basket {
2
3    Fruit orange = new Fruit("Orange", "orange");
4    Fruit lemon = new Fruit("Lemon", "yellow");
5    Fruit mango = new Fruit("Mango", "orange");
6    public final List<Fruit> fruits = Arrays.asList(orange, lemon, mango
         );
7
8 }
```

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:

```
1 void withIterator() {
2    Basket basket = new Basket();
3    Iterator<Fruit> iterator = basket.fruits.iterator();
4    while (iterator.hasNext()) {
5       Fruit current = iterator.next();
6       if (current.color.equals("orange")) {
7          System.out.println(current.name);
8       }
9    }
10 }
```

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 then write down it's name. Now lets look at how we can do this in Java 8 making use of Streams.

```
1 void withStreams() {
2    Basket basket = new Basket();
3    basket.fruits.stream()
4                 .filter(current -> current.color.equals("orange"))
5                 .forEach(current -> System.out.println(current.name));
6 }
```

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 want to write down it's name. This approach lets us specify our intentions clearly which in turn makes the code easier to read and understand.

With that said, using just the Stream API Java 8 provides, we soon come to a point where we would like to create a new operation, to that end Java provides the Spliterator interface which we can implement or the AbstractSpliterator class that we can extend.

```
1  public class Example {
2   static <T> Stream <T> filterOdd (Stream <T> source ) {
3    return StreamSupport.stream(new OddFilter<>(source.spliterator()),
        false);
4   }
5   static class OddFilter <T> extends AbstractSpliterator <T> {
6    final Consumer <T> doNothing = item -> {};
7    final Spliterator <T> source ;
8    public Odd (Spliterator <T> source ) {
9     super (odd(source.estimateSize()), source.characteristics());
10    this.source = source;
11   }
12   @Override
13   public boolean tryAdvance(Consumer <? super T> action ) {
14    if (!source.tryAdvance(doNothing)) return false ;
15     return source.tryAdvance(action);
16   }
17   private static long odd( long l) {
18    return l == Long.MAX_VALUE ? l : (l +1)/2;
19   }
20  }
```

```
21  public static void main(String[] args) {
22    filterOdd(Stream.of(new Integer[]{1, 2, 3, 4})
23          .map(i -> i + 4))
24          .forEach(System.out::println);
25  }
26 }
```

Listing 1.5: Spliterator example

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.

This approach is verbose, and does not benefit from bulk traversing, as stated by the Spliterator documentation, the forEachRemaining[**?**] 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. To overcome these shortcomings 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, i.e. if the user expresses how to traverse a sequence item by item we will generate a function that will allow for bulk traversing and vice-versa.

1.2 Query

A Query represents a sequence of elements supporting sequential operations in Jayield. This class implements all the utility methods with the same name, semantics and behaviour as those provided by Java's Stream. The traversal methods are specified by two distinct interfaces, Advancer which allows the user to express how to traverse a sequence item by item and Traverser that is used to express how to bulk traverse a sequence. Both of these are functional Interfaces and both use a third functional interface to process through elements, Yield. Yield is equivalent to the Java Consumer, however, to reinforce the role of this functional interface as the element generator, it was decided to create a new one.

```
1 interface Traverser <T> { void traverse (Yield <T> yield );}
2 interface Advancer <T> { boolean tryAdvance (Yield <T> yield );}
3 interface Yield <T> { void ret(T item );}
```

Listing 1.6: Interfaces

Query provides a way for the user to define new operations through the then function.

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

Listing 1.7: Traversal function generator

This method has a function parameter "next", which receives the previous (downstream) sequence and returns a new implementation of a traversal function, a Traverser.

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

      boolean tryAdvance(Yield<T> yield);

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

      static <U> Advancer<U> from(Traverser<U> source) {
         return Generator.generateAdvancer(source);
      }

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

      default Traverser<T> traverser() {
         return yield -> {
            while(this.tryAdvance(yield));
         };
      }


   }
```

Listing 1.8: Advancer

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 1.8 listing.

Going back to the filterOdd example, if we wanted to implement it we would do it in one of the following ways.

```
public class App{
  static <U> Traverser <U> filterOdd (Query<U> prev) {
    return yield -> {
      final boolean [] isOdd = { false };
      prev.traverse(item -> {
        if(isOdd[0]) yield.ret(item);
        isOdd[0] = !isOdd[0];
      });
    };
  }

  public static void main(String[] args) {
    Query.of(1, 2, 3, 4)
          .map(i -> i + 4))
          .then(App::filterOdd)
          .forEach(System.out::println);
  }
}
```

Listing 1.9: filterOdd using Query and Traverser

```
public class App{
  static <U> Advancer <U> filterOdd (Advancer<U> prev ) {
    return yield -> {
      if (!prev.tryAdvance(item -> {})) return false;
        return prev.tryAdvance(yield);
    };
  }

  public static void main(String[] args) {
      Query<Integer> query = Query.of(1, 2, 3, 4)
                           .map(i -> i + 4);
      Advancer<Integer> filterOdd = filterOdd(Advancer.from(query));
      while (filterOdd.tryAdvance(System.out::println));
  }
}
```

Listing 1.10: filterOdd using Advancer

This approach is less verbose, does not break fluent query invocation and if we call then we will be bulk traversing efficiently without needing to define how to traverse item by item.

1.3 Use Case

When we first started developing this project, it was though that generating an Advancer for every Traverser would be best, this was mainly due to short circuiting operations, meaning, operations that do not traverse the whole sequence, such as the operation limit(int n) that stops once n number of elements have been processed. As we discovered, generating the Advancer is not always necessary, we can short circuit operations through the use of lightweight exceptions, which are exceptions that are not as disruptive to the process as a regular exception. With that said, generating Advancers is still important for operations that require the use of more than one sequence such as zip, which creates a sequence that combines the elements of multiple sequences.

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.

1.4 Approaches

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:

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

Listing 1.11: Map Traverser

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

```
<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 1.12: 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.

To solve this issue, two paths were found[**?**], the use of SerializedLambda[**?**] or the use of another library called Bytebuddy.

1.5 SerializedLambda

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 static method that was generated for this lambda as well as get the captured arguments. Being able to reach the lambda's method would allowed us to instrument it, to do so, we used 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:

```
1 final BoolBox elementFound = new BoolBox();
2 Method generatedTryAdvance = getTryAdvance(...);
3 Advancer generated = y -> {
4   elementFound.reset();
5   Yield<R> wrapper = wr -> {
6     y.ret(wr);
7     elementFound.set();
8   };
9   arguments[arguments.length - 1] = wrapper;
```

```
10  while(elementFound.isFalse() && (Boolean) generatedTryAdvance.invoke(
        newClass, arguments));
11  return elementFound.isTrue();
12 };
```

Listing 1.13: 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.

A downside to this approach is that it would mean the Traverser functional interface would need to extend Serializable, which would be otherwise unnecessary.

1.6 Bytebuddy

This approach was not as thoroughly investigated as the previous one, with that said, the main difference between the two is the way the class of the lambda and the arguments are obtained. To use this approach we need to add the ByteBuddyAgent to the class path and to call the install static method it provides to get a hold of it's implementation of the Instrumentation Interface. We are then able to obtain the class of the Traverser we are trying to instrument, and this is where the main difference resides, we get the class directly not a method like in the SerializedLambda approach. This class has the fields, that the lambda captured, and an implementation of the traverse method. Which makes it seem like a similar approach to the SerializedLambda is possible, we would just make use of a custom ClassVisitor that would generate an Advancer from the Traverser implementation and then return it.