INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Extensão eficiente da API de java.util.stream

DIOGO RAFAEL ESTEVES POEIRA

Licenciado em Engenharia Informática e de Computadores

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

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Janeiro, 2018

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Extensão eficiente da API de java.util.stream

DIOGO RAFAEL ESTEVES POEIRA

Licenciado em Engenharia Informática e de Computadores

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

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Janeiro, 2018

1 Introduction

In this project we will analyse the definition of an alternative to the java.util.stream API that will allow us overcome its limitations as well as being interoperable with it.

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 sequence of data by providing methods that accept behaviour as parameters, allowing the programmers to express what their intentions are when calling the method. However using just the Stream API Java 8 provides, we soon come to a point where we would like to create a new operation, to do so Java provides the Spliterator interface which we can implement or the AbstractSpliterator class that we can extend.

```
public class App{
  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 ;
    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);
    }

    private static long odd( long l) {
      return l == Long.MAX_VALUE ? l : (l +1)/2;
    }
  }

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

Listing 1.1: Spliterator Example

This approach is verbose and breaks the fluent query invocation, as well as not benefiting from bulk traversing, seeing as forEachRemaining[**?**] method just calls tryAdvance multiple times if not overridden, having to define both tryAdvance and forEachRemaining for efficient bulk traversing. To overcome these shortcomings we will implement a solution that allows the user to define a single way of traversing 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 Series

Series is the provided implementation of a sequential type. This class implements all the utility methods with the same name, semantics and behaviour as those provided by Java's Stream. However Series is a class instead of an interface with final methods instead of default ones. 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 Traversable 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.

```
interface Traversable <T> { void traverse (Yield <T> yield );}
interface Advancer <T> { boolean tryAdvance (Yield <T> yield );}
interface Yield <T> { void ret(T item );}
```

Listing 1.2: Interfaces

Series provides two methods that allow the user to define new operations, these are, traverseWith and advanceWith.

```
<R> Series<R> traverseWith(Function<Series<T>, Traversable<R>> then)
<R> Series<R> advanceWith(Function<Series<T>, Advancer<R>> then)
```

Listing 1.3: Traversal function generators

These methods have a function parameter "then", which receives the previous (downstream) sequence and returns a new implementation of a traversal function: Traversable or Advancer. The missing implementation will be inferred from the available one, so if you call traverseWith, we will have a Traversable available and will infer an Advancer.

So to implement a filterOdd operation we would do it in one of the following ways.

```java
public class App{
  static <U> Traversable <U> filterOdd (Series<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) {
    Series.of(new Integer[]{1, 2, 3, 4})
            .map(i -> i + 4))
            .traverseWith(App::filterOdd)
            .forEach(System.out::println);
  }
}
```

Listing 1.4: filterOdd using traverseWith

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

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

Listing 1.5: filterOdd using advanceWith

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

1.3 Use Case

The use case that will be analysed, developed and tested will be the use of traverseWith as well as the generation of the missing traverser, in other words, the Advancer.

The difficulty with this use case is the fact that a Traversable 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 Traversable's code and instrument the required parts to generate a new Advancer. To test if the generated Advancer is working properly we will chain the traverseWith call with a short-circuiting operation to verify that the sequence is not fully traversed.

1.4 Approaches

To achieve the expected result a pseudo-solution was thought of assuming the Traversable follows these pre-conditions:

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

Assuming 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 Traversable:

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

Listing 1.6: Map Traversable

A valid definition of a map Advancer would be:

```
<T, R> Traversable<R> map(Series<T> source, Function<T, R> mapper) {
  (*@\textcolor{blue}{boolean[] noElementFound = new boolean[] \{true
     \};}@*)
  return yield -> {
    (*@\textcolor{blue}{noElementFound[0] = true;}@*)
    while(noElementFound[0] && source.tryAdvance(e -> {
        yield.ret(mapper.apply(e)));
        (*@\textcolor{blue}{noElementFound[0] = false;}@*)
      });
    (*@\textcolor{blue}{return noElementFound[0] == false;}@*)
  };
}
```

Listing 1.7: Map Traversable

With that said, there are corner cases that do not follow the pre-conditions as is the case of the flatMap operation which does not call yield.ret(...), these cases need further studying.

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 Traversable. The problem is, if the Traversable 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 Traversable'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:

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

Listing 1.8: 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 Traversable 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 Traversable 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, declared 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 Traversable implementation and then return it.