# Java Stream Fusion: Adapting FP mechanisms for an OO setting

Francisco Ribeiro
francisco.j.ribeiro@inesctec.pt
HASLab/INESC TEC
Universidade do Minho
Braga, Portugal

João Saraiva
saraiva@di.uminho.pt
HASLab/INESC TEC
Universidade do Minho
Braga, Portugal

Alberto Pardo
pardo@fing.edu.uy
Instituto de Computación
Universidad de la República
Montevideo, Uruguay

## ABSTRACT

In this paper, we show how *stream fusion*, a program transformation technique used in functional programming, can be adapted for an Object-Oriented setting. This makes it possible to have more Stream operators than the ones currently provided by the Java Stream API. The addition of more operators allows for a greater deal of expressiveness. To this extent, we show how these operators are incorporated in the *stream* setting.

Furthermore, we also demonstrate how a specific set of optimizations eliminates overheads and produces equivalent code in the form of *for loops*. In this way, programmers are relieved from the burden of writing code in such a cumbersome style, thus allowing for a more declarative and intuitive programming approach.

## CCS CONCEPTS

• **Software and its engineering** → *Object oriented languages*; *Recursion*; *Software libraries and repositories.*

## KEYWORDS

Object-Oriented Programming, Functional Programming, Program Fusion

**ACM Reference Format:**
Francisco Ribeiro, João Saraiva, and Alberto Pardo. 2019. Java Stream Fusion: Adapting FP mechanisms for an OO setting. In *XXIII Brazilian Symposium on Programming Languages (SBLP 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3355378.3355386

## 1 INTRODUCTION

Over the last several years, programming languages have evolved in order to provide powerful abstractions to programmers. Examples of such abstractions are models which represent code abstractions, powerful type systems and recursion patterns allowing the definition of functions that abstract the data type they traverse.

Recently, Java 8 adopted lambda expressions and streams to improve the language expressivity. Lambda expressions allow us to write functions that do not belong to any specific class and pass them as a parameter just like primitive data types and Objects. Streams can be chained in order to represent a pipeline of operations. Each stream represents one operation. Intermediate operations are lazy, i.e. they only return a stream describing what operation is to be executed. Terminal operations trigger the execution of a pipeline, thus producing the desired result. Combining these two elements provides a powerful mechanism to manipulate Java collections. However, as reported in [11], expressivity and performance are still lacking. In fact, well-optimized Java 8 streams do not support important operators, like for example the zip operator, and are still an order of magnitude slower than hand-written loops [11].

Lambda expressions are widely used in the context of functional programming. Together with the use of high-order functions, such as *map* and *filter*, they make it possible to express non-trivial sequences of actions with little effort. However, the execution of these recursion patterns has several efficiency problems, either by doing more traversals than necessary, or by creating intermediate data structures. Thus, a lot of work has been developed for an efficient execution of these mechanisms. Techniques such as shortcut fusion [2, 6, 7, 13, 15, 16], program tupling [9], or deforestation [12, 17], among others, provide program optimizations which eliminate the overhead introduced by expressing algorithms in this higher-order functional style of programming: different traversals are fused into a single one, and intermediate data structures are eliminated.

In general, program fusion aims to eliminate data structures which are produced and consumed right away. Fusion is based on the fact that programs written in a compositional style can be rewritten in a more efficient way, avoiding the creation of intermediate structures. For example, the concise Haskell function `all`,

```
all p xs = and (map p xs)
```

checks if all elements of a list `xs` satisfy a given predicate `p`. As we can see, it is expressed as a composition of functions `and` (conjunction of a list of booleans) and `map`. The `and` function is a fold on lists and therefore `all` is the composition of two higher order functions. (For reasons that will be clearer later, we use the definition of `and` in terms of foldl.)

```
all p xs = foldl (&&) True (map p xs)
```

In this definition, however, an intermediate list is created to communicate the results from one function to another. However, this program can be rewritten in a way which does not make use of an intermediate list.

```
all' p xs = h True xs
  where
    h b [] = b
    h b (x:xs) = h (b && p x) xs
```

Although this version is more efficient, we lose readability and conciseness. Furthermore, one does not wish to write programs in this style and, instead, prefers to use a more compositional style such as the first version of `all` provided but without performance


*SBLP 2019, September 23–27, 2019, Salvador, Brazil*

ACM ISBN 978-1-4503-7638-9/19/09…$15.00
https://doi.org/10.1145/3355378.3355386

penalties. Therefore, we want compilers to be able to automatically perform this optimization. The Haskell compiler offers a mechanism to define rewrite rules, stating e.g. how to fuse functions together. An example rule would be:

```
map f (map g xs) = map (f.g) xs
```

However, this approach has a key drawback: it requires a rule for each possibile higher-order function combination. In other words, we need to define a specific compiler optimization for each combination. In order to avoid the case-by-case definition of all such program fusion optimizations, Coutts *et al.* [1] proposed functional *stream fusion* as a generic program transformation technique where functions are fused without the need to explicitly state/implement the rules performing those transformations.

In this paper we model functional stream fusion [1] in the Java Object Oriented Programming (OOP) language in the form of a library called *FStream*[1]. We express the stream fusion data type as a set of Java classes and their constructors. Then, we model in this OO setting the higher-order recursion patterns *map*, *filter* and *foldl* offered by standard Java Streams. Moreover, we extend such recursion patterns with other widely used functional patterns such as *zip* and *foldr* which are not supported by Java 8, thus limiting their expressiveness. Finally, we use stream fusion to optimize Java methods expressed as streams.

Additionally, we also discuss refactorings to fuse higher-order patterns into loops. That is to say, that we refactor lambda expressions and higher-order functions expressed in our functional stream setting into regular Java *foreach* loops, as opposed to [4]. More precisely, our framework transforms a Java stream version of `all` into a loop variant of `all'`.

The paper is organized as follows: Section 2 briefly presents functional stream fusion, and in Section 3 we show how to express it in Java. Section 4 explains how we integrated *Continuation Passing Style* into the setting's implementation. Section 5 presents a set of Java optimizations that eliminate overhead introduced by stream fusion. In Section 6 we present transformations to simplify the obtained loops. Finally, Section 7 gives our conclusions.

## 2 STREAM FUSION

*Stream Fusion* [1] is an automatic deforestation system that takes a different approach compared to more traditional fusion systems. In *Stream Fusion*, the operations over the original list structure are transformed in order to work over an alternative, co-inductive representation of the list that captures its generation process. As Coutts et al. [1] state, the natural operation over a list is a *fold*, while on the other hand, the natural operation over a stream is an *unfold*. Therefore, a list's co-structure is a stream.

The `Stream` datatype encloses that unfolding behaviour. In order to achieve this, it wraps an initial state and a stepper function which specifies how elements are produced from the stream's state.

```
data Stream a = ∃s. Stream (s → Step a s) s
```

The stepper function produces a `Step` element, which permits three possibilities:

[1]https://github.com/FranciscoRibeiro/FStream

```
data Step a s = Done
              | Yield a s
              | Skip s
```

The `Step` datatype allows the co-structure to be non-recursive, thanks to the `Skip` data constructor. This is the key point of the stream fusion system. The `Skip` constructor is what allows the production of a new state without yielding a particular element. This is a crucial point as it permits every stepper function to be non-recursive. The `Done` and `Yield` alternatives are quite simple as they pinpoint the end of a stream and carry an actual element together with a reference to the rest of the stream's state, respectively.

To convert lists to streams and vice-versa, we need two functions.

```
stream :: [a] → Stream a
stream xs0 = Stream next xs0
  where
    next [] = Done
    next (x : xs) = Yield x xs

unstream :: Stream a → [a]
unstream (Stream next0 s0) = unfold s0
  where
    unfold s = case next0 s of
      Done → []
      Skip s' → unfold s'
      Yield x s' → x : unfold s'
```

The function `stream` creates a Stream with:

- a stepper function `next` which is non-recursive and yields each element of the stream as it unfolds;
- a state, which consists of the list itself.

The function `unstream` creates a list by unfolding the given stream, repeatedly calling the stream's stepper function.

Implementing a function over streams is quite simple; given an input stream, one has to define the initial state and the particular stepper function for the stream to be returned as result.

*Map* and *filter* are two well known higher order functions which have implementations over streams: `map f xs` applies function `f` to each element of `xs` whereas `filter p xs` returns a list with all the elements of `xs` that fulfill the condition stated by `p`. The implementation of *map* over streams is the following:

```
map_s :: (a → b) → Stream a → Stream b
map_s f (Stream next_0 s_0) = Stream next s_0
  where
    next s = case next_0 s of
      Done → Done
      Skip s' → Skip s'
      Yield x s' → Yield (f x) s'
```

Function `filter` is the one where one can appreciate the advantage of having a non-recursive stepper function. This non-recursive implementation can only be achieved due to the `Skip` element because it avoids the need to recursively go through the structure to find elements satisfying the predicate. As a result, the Haskell compiler can optimize the code due to the absence of recursion.

```
filter_s :: (a → Bool) → Stream a → Stream a
filter_s p (Stream next_0 s_0) = Stream next s_0
  where
    next s = case next_0 s of
      Done → Done
      Skip s' → Skip s'
      Yield x s' | p x → Yield x s'
                 | otherwise → Skip s'
```

The introductory example shown in Section 1 makes use of function `foldl` over lists. Indeed, this function is also present in the *Stream Fusion* setting.

```
foldl_s :: (b → a → b) → b → Stream a → b
foldl_s f z (Stream next s_0) = go z s_0
  where
    go z s = case next s of
          Done → z
          Skip s' → go z s'
          Yield x s' → go (f z x) s'
```

List functions can then be written in terms of their stream counterparts by using `stream` and `unstream`.

```
map f = unstream . map_s f . stream
filter p = unstream . filter_s p . stream
foldl f z = foldl_s f z . stream
```

Recall that function `all`, presented in Section 1, is a composition of `foldl` and `map`. Therefore, in terms of streams it is expressed as:

```
all p = foldl_s (&&) True . stream . unstream . map_s p . stream
```

In the *Stream Fusion* setting, the intermediate list that is being created in the composition is then removed by the application of the following law:

```
stream . unstream = id
```

which is implemented as a GHC rewrite rule. As result, function `all` is written as follows:

```
all p = foldl_s (&&) True . map_s p . stream
```

In [1], program tranformation rules of GHC are used in order to completely fuse the codes of $\texttt{fold}_s$ and $\texttt{map}_s$. Those transformations exploit the fact that their stepper functions are non-recursive. Next, we show how to model streams in Java, how to write the stream-based higher order functions/methods, and, finally how to mimic such program transformations as Java refactorings.

## 3 STREAM FUSION IN JAVA

In order to model functional streams in Java, first we need to express the `Stream` and `Step` datatypes as Java classes.

### 3.1 Stream and Step classes

As one could previously see, the Stream datatype encapsulates a stepper function `(s → Step a s)` and a state represented by `s`. In order to represent this datatype in Java, a class called `FStream` (standing for *Fusion Stream*) was created.

```
public class FStream<T>{
  public Function<Object, Step> stepper;
  public Object state;
  ...
```

As Haskell is a polymorphic language, the datatype `Stream` is defined in a generic way. More precisely, `a` and `s` are type variables, which means they can be of any type, and `Stream` and `Step` are parameterized types.

To achieve this in Java, we use *generics*, which allows us to use types as parameters when defining classes. In this case, `T` is a type parameter of the class `FStream`, meaning that a stream can hold values of any type (`Integer`, `String`, etc.), just as its Haskell counterpart.

Resorting to the `Function` class introduced in Java 8, it is easy to store the desired behaviour for the stepper function in an instance variable. The input for this stepper function is a state (`Object`) and the output is a `Step` object. There are different possibilities for the type of the state encapsulated by the stream. As such, the more generic `Object` class is used.

Another important datatype in this implementation, and the main responsible for the advantages that the stream approach allows, is `Step`. In Haskell, its definition has three value constructors: `Done`, `Yield` and `Skip`. Similarly to `Stream`, the datatype `Step` is defined in a polymorphic way and so the same approach using *generics* was taken.

```
public abstract class Step<T,S>{
  public T elem;
  public S state;
```

To reflect the role of each of the value constructors, a separate class was created for each of them. Each of these classes is a specialization of `Step` and, as such, they extend that class.

`Done` represents the end of a stream. When an object of this type is detected, we know we have reached the end of the stream. As this object does not carry any particular value, its implementation is quite simple.

```
public class Done extends Step{}
```

A `Yield` object carries an actual element and the rest of the state coming after the element in question. It has two types as parameters, as previously seen with other classes, which are in conformity with the generic types of the corresponding stream.

```
public class Yield<T,S> extends Step{
 public Yield(T e, S s){
  this.elem = e;
  this.state = s;
```

Finally, we define the `Skip` class. Although this element is not important to understand the approach being presented, it is of extreme importance in the implementation because it is what enables the stepper functions to be non-recursive and thus allowing fusion, which is the mechanism behind all the optimization and, consequently, the efficiency improvements.

```
public class Skip<S> extends Step{
  public Skip(S s){ this.state = s; }
```

### 3.2 Methods

After creating the necessary functional datatypes as Java classes, we need to generate an `FStream` object from a known state. To simplify our presentation, we consider the state to be a list (more general types are discussed in [14]).

In Haskell, this is represented by the function `stream` (presented in Section 2). The state of the stream is the list itself and elements are yielded one at a time as the stream gets traversed (unfolded).

To recreate this in Java, a method in the `FStream` class called `fstream` was implemented. The return type of the method is an `FStream` of the same type of objects ($T$) as the input list. The stepper function, as seen in the Haskell implementation, returns a `Done` object if the list is empty, meaning it reached the end of the list. Otherwise, it returns a `Yield` object yielding the first element of the list and the rest of that list as the remaining state. Note that the implementation of the stepper function is saved inside a variable of type *Function* (*nextStream*). The returned *FStream* holds it and this stepper function is only executed when its method *apply* is called (as one shall see later).

```
public static <T> FStream<T> fstream(List<T> list){
  Function<Object, Step> nextStream = x -> {
    List aux = (List) x;
    if(aux.isEmpty()) return new Done();
    else{
      List<T> sub = aux.subList(1, aux.size());
      return new Yield<T, List<T>>((T) aux.get(0), sub);
    }
  };
  return new FStream<T>(nextStream, list);
```

In order to create a list back from a stream, the `unstream` function repeatedly calls the stepper function of the stream, unfolding it. The Haskell implementation was shown in Section 2. The most important part of this function is the `unfold`. In the equivalent Java method that we define, this unfolding behaviour is implemented in the form of a `while` loop, since it is a tail recursive function, and, as a consequence, efficiently expressed by a loop. The stream's stepper function is repeatedly called inside the loop. Depending on the result of that method call, a different behaviour can occur:

- First of all, if the object returned by the stepper function is a `Done`, then the loop finishes, as the end of the stream has been reached.
- If the object is of type *Skip*, then the unfolding process continues with the rest of the state.
- Otherwise, if it is a `Yield` object, it will add the yielded element to the result list and continue unfolding the stream.

```
public List<T> unfstream(){
  ArrayList<T> res = new ArrayList<>();
  Object auxState = this.state;
  boolean over = false;

  while (!over) {
    Step step = this.stepper.apply(auxState);
    if (step instanceof Done) over = true;
    else if (step instanceof Skip) auxState = step.state;
    else if (step instanceof Yield) {
      res.add((T) step.elem);
      auxState = step.state;
    }
  }
  return res;
```

Higher order functions are, perhaps, the most important aspect of functional programming [10]. They are a powerful mechanism that allow programmers to define what the computations *are* instead of programming the steps that compose the computation.

As shown in Section 2, the *map* higher order function takes a function as a parameter. In the Java implementation, that parameter function receives an input of type `T` and its output is of type `S`. Therefore, the `FStream` object returned by the method `mapfs` is a stream holding objects of type `S`, which is the return type of the function to be applied to each of the stream's elements. As any other stream, the stream being created needs a stepper function of its own. The behaviour for that function is saved in the variable `nextMap` and it produces a different `Step` object depending on the outcome of the `FStream`'s stepper function to which we are applying `mapfs`.

`Done` represents the end of the stream. So, if it is encountered, then an object of that type will also be created. If a `Skip` object was produced, it means that that element should not be dealt with. As such, another `Skip` is created referencing the rest of the stream's state. Finally, if an actual element is being yielded, i.e. a `Yield` object exists, then the input function for `mapfs` (*funcTtoS*) is applied to that element and the result is placed into the new `Yield` that is going to be instantiated, along with the rest of the stream's state.

```
public <S> FStream<S> mapfs(Function<T,S> funcTtoS){
  Function<Object, Step> nextMap = x -> {
    Step aux = this.stepper.apply(x);
    if(aux instanceof Done) return new Done();
    else if(aux instanceof Skip) return new Skip<>(aux.state);
    else if(aux instanceof Yield)
      return new Yield<>(funcTtoS.apply((T)aux.elem), aux.state);
    return null;
  };
  return new FStream<S>(nextMap, this.state);
```

The `filterfs` method (whose equivalent Haskell implementation has been presented in Section 2) is similar to the previous `mapfs` method, although it takes a `Predicate` as a parameter instead of a `Function`. Its stepper function's behaviour is quite similar to the one explained before in `mapfs`.

The only difference lies in the case when the stepper function of the stream being filtered returns a `Yield` object. In that situation, the filter's stepper function (*nextFilter*) should evaluate if the element yielded satisfies the given predicate. If it does, then a new `Yield` is created carrying the element in question and the rest of the stream's state. Otherwise, as the element does not satisfy the predicate, it should not be present in the resulting stream and, therefore, a `Skip` object is created with a state containing the subsequent elements.

```
public FStream<T> filterfs(Predicate p){
  Function<Object, Step> nextFilter = x -> {
    Step aux = this.stepper.apply(x);
    if(aux instanceof Done) return new Done();
    else if(aux instanceof Skip) return new Skip<>(aux.state);
    else if(aux instanceof Yield){
      if(p.test(aux.elem))
        return new Yield<>((T) aux.elem, aux.state);
      else return new Skip<>(aux.state);
    }
    return null;
  };
  return new FStream<T>(nextFilter, this.state);
```

In the *Stream Fusion* approach, there are two other terminal operations besides `unstream`. Those operations are `foldr` and `foldl`.

As seen in Section 2, `unstream` and `foldl` are implemented in a recursive way. In contrast, in the Java implementation recursion is converted to loops whenever possible. That is the case with `unstream` and `foldl`, but not for `foldr` as it is not *tail recursive*. Because of that it cannot be expressed in terms of a loop in the same manner the other two functions are. This is discussed in Section 4.

The `go` function in $\texttt{foldl}_s$ (presented in Section 2) is implemented in Java as a *while* loop. On every iteration, the stepper function of the input stream is called. If the resulting `Step` happens to yield an element, function `f` is applied to the current value and to the element in question. The state moves forward whenever `step` evaluates to a `Skip` or a `Yield`.

```
public <S> S foldl(BiFunction<S,T,S> f, S value) {
  Object auxState = this.state;
  boolean over = false;

  while (!over) {
    Step step = stepper.apply(auxState);
    if (step instanceof Done) over = true;
    else if (step instanceof Skip) auxState = step.state;
    else if (step instanceof Yield) {
      auxState = step.state;
      value = f.apply(value, (T) step.elem);
    }
  }
  return value;
```

Let us now return to our motivating example presented in Section 1: using the `FStream` setting, the higher-order `all` function is written as follows:

```
Boolean res = fstream(xs).mapfs(p).foldl((x,y) -> x && y, true);
```

In the end of Section 6.1, an equivalent version using a *for each* loop is presented.

## 3.3 Extending patterns

While *Java Streams* currently support several functional operators, their expressiveness is still lacking. Let us assume that a polynomial expression like $3x^2 + 5x + 1$ is represented by a list of its coefficients in reverse order: [1,5,3], where the list index is the corresponding power. For a given value `x` and a list `cs` representing a polynomial, one can write a Haskell function calculating the result through the composition of `zip`, that explicitly builds the list of pairs $(coef, power)$ - $[(1,0),(5,1),(3,2)]$ in our example, and `foldr` in order to perform the accumulation of the addition of all individual terms.

```
pol x cs =
 foldr (\(a,b) acc -> acc + a*(x^b)) 0 $ zip cs [0..length(cs)-1]
```

In fact, *zip* and $foldr$ are widely used to express inumerous functions in a functional setting. Unfortunately, the *right fold* and *zip* operators are not present in *Java Streams*, and as a consequence we can not express such functions in functional Java. Moreover, it is not simple to extend the Java Streams library to efficiently support more higher-order operators [11]. Next, we show how we can easily express a *right fold* and a *zip* pattern in our setting.

*3.3.1 Right fold.* As previously stated, `foldr` is not *tail recursive* and its conversion to a loop in the Java implementation needs to be somewhat different from the one regarding `unstream` and `foldl`.

```
foldr_S :: (a → b → b) → b → Stream a → b
foldr_S f z (Stream next s_0) = go s_0
  where
    go s = case next s of
        Done → z
        Skip s' → go s'
        Yield x s' → f x (go s')
```

Different alternatives for an implementation of `foldr` are discussed in [14].

In order to come up with an efficient implementation, one can resort to a particular mechanism in the scope of functional programming, *Continuation Passing Style* (CPS) [8]. With CPS, one explicitly passes the control of the operation in the form of a so called *continuation*. This is useful, as it allows us to create an implementation for *foldr* that is *tail recursive*.

```
foldr' f z [] cont = cont z
foldr' f z (x:xs) cont = foldr' f z xs (\a -> cont (f x a))
```

As a result, an equivalent implementation in the form of a loop can be created for *foldr'*.

```
public <S> S foldrTailRec(BiFunction<T,S,S> f, S value){
  Continuation.b = f;
  Continuation cont = new ContinuationId();
  boolean over = false;
  Continuation.globalState = this.state;
  Continuation.res = value;

  while(!over){
    Step step = this.stepper.apply(Continuation.globalState);
    if(step instanceof Done){
      cont = cont.execute(Continuation.res);
      if(cont == null){ over = true; }
    } else if(step instanceof Skip){
      Continuation.globalState = step.state;
    } else if(step instanceof Yield){
      Continuation.globalState = step.state;
      Continuation<S, ContinuationListElem<T,S>> nextCont = new
     ContinuationListElem<T,S>((T) step.elem, cont);
      cont = new ContinuationListFold(nextCont);
    }
  }
  return (S) Continuation.res;
```

Continuations are discussed in more detail in Section 4.

*3.3.2 Zip.* When a function consumes more than one stream at the same time, some extra care is needed in order to handle the stream's state. The $zip_S$ function is an example of such a function. This function receives two lists as input and produces a list of corresponding pairs. In case the two input lists are of different lengths, the remaining elements of the longer list are discarded. The fact that this function has to deal with `Skip`, means that the function might be able to extract one element from one of the streams at some point but might not be able to do it simultaneously from the second stream.

If that is the case, the extracted element is saved inside the current state and we iterate over the second stream until an element gets extracted too.

For that, the `Maybe` datatype is used. When no element is being carried, the value is empty (`Nothing`). When an element is yielded and waiting to be coupled with another one, the value is saved inside `Just`.

```
zip_S :: Stream a → Stream b → Stream (a, b)
zip_S (Stream next_a s_a0 ) (Stream next_b s_b0 ) =
  Stream next (s_a0, s_b0, Nothing)
    where
      next (s_a, s_b, Nothing) =
        case next_a s_a of
          Done → Done
          Skip s'_a → Skip (s'_a, s_b, Nothing)
          Yield a s'_a → Skip (s'_a, s_b, Just a)
      next (s'_a, s_b, Just a) =
        case next_b s_b of
          Done → Done
          Skip s'_b → Skip (s'_a, s'_b, Just a)
          Yield b s'_b → Yield (a, b) (s'_a, s'_b, Nothing)
```

In order to create an equivalent Java method, the `Optional` class is used. Similarly to Haskell's `Maybe`, an `Optional` object can either have a value present or be empty.

The state produced and handled by `zipfs`'s stepper function consists of a tuple of size three. Therefore, a class `Triple` was created in order to represent values of the form *(stateA, stateB, Optional)*.

The objects that the returned stream represents consist of tuples too, although these ones have size two. An identical approach was taken in order to represent these values. As such, the `Pair` class was created.

The first action the stepper function needs to perform is to check if the `Optional` element inside the input state is empty or not.

After that, the same procedure of all the other stepper functions so far is followed. Depending on the type of the `Step` returned, `nextZip` performs the appropriate operation.

When unfolding the first stream (`Optional` is empty), a value is saved only when a `Yield` is encountered. Therefore, this results in a value being present inside the `Optional` object, which makes the

stepper function unfold the second stream, beginning the search for an element to complete the `Pair`. Thus, the `Optional` value inside the `Triple` is only set to empty again when another `Yield` is found.

```
public <S> FStream<Pair<T,S>> zipfs(FStream<S> streamB){
  Function<Object, Step> nextZip = x -> {
    if(!(((Triple) x).getElem()).isPresent()){
      Step aux = this.stepper.apply(((Triple) x).getStateA());
      if(aux instanceof Done) return new Done();
      else if(aux instanceof Skip)
        return new Skip<>(new Triple(aux.state, ((Triple) x).
     getStateB(), Optional.empty()));
      else if(aux instanceof Yield)
        return new Skip<>(new Triple(aux.state, ((Triple) x).
     getStateB(), Optional.of(aux.elem)));
    }
    else{ //There is a value present in Optional
      Step aux = streamB.stepper.apply(((Triple) x).getStateB());
      if(aux instanceof Done) return new Done();
      else if(aux instanceof Skip){
        return new Skip<>(new Triple(((Triple) x).getStateA(),
     aux.state, ((Triple) x).getElem()));
      else if(aux instanceof Yield)
        return new Yield<>(new Pair<>(((Triple) x).getElem().get
     (), aux.elem), new Triple(((Triple) x).getStateA(), aux.
     state, Optional.empty()));
    }
    return null;
  };
  return new FStream<>(nextZip, new Triple<>(this.state, streamB.
     state, Optional.empty()));
```

Let us rewind to the polynomial function previously presented. In the Haskell implementation provided in the beginning of this section, in order to create the list of numbers representing the *powers* we use *ranges*: `[0..length(cs)-1]`. In a strict setting like Java, we would have to explicitly generate this list from the coefficients. However, we can use streams to come up with an implementation that is also based on *laziness*. Methods `unfoldr` and `take` from the *FStream* setting allow us to emulate the desired behaviour.

```
Function<Integer, Optional<Pair<Integer,Integer>>> builder =
 v -> Optional.of(new Pair<>(v, v+1));
BiFunction<Pair<Integer,Integer>, Integer, Integer> f =
 (pair,acc) -> (int) (acc+pair.getX()*pow(x, pair.getY()));

FStream<Integer> powers = unfoldr(builder, 0).take(cs.size());
return fstream(cs).zipfs(powers).foldrTailRec(f, 0);
```

As one can see, because the *FStream* setting supports both `zip` and `foldr`, it allows for an elegant representation of the example in question.

## 4 CONTINUATIONS

Different `Continuations` are going to be needed for different reasons during execution. Yet, they all still share one common blueprint. Therefore, a more general class was created. All other, more specific, `Continuations` will extend this super class.

```
public abstract class Continuation<S, R extends Continuation<S,R
     >> {
public static Object globalState;
public static Object res;
public static BiFunction b;
public Continuation<S,R> nextCont;

public abstract Continuation execute(S value);
```

The class signature forces every subclass to have its type parameters be the same as the superclass ones. All objects of type `Continuation` will share three class variables:

- **globalState:** the current stream state as it gets modified;
- **res:** the accumulator value that is built throughout the folding operation;
- **b:** the function that is repeatedly applied to every element retrieved from the stream and the accumulator value.

As the folding operation progresses through the stream, each `Continuation` that is created along the way holds a reference to the next one - `nextCont` - i.e. what to do next (as this style of programming implies). As a result, a chain of `Continuation` objects will have been created by the time the fold operation finishes unrolling the stream. An object belonging to this class represents an action. As such, every object needs to define a method `execute` which indicates what should be performed.

`foldr` consumes a list from right to left. However, elements are retrieved from a stream from its beginning to its end, i.e. from left to right. As a result, as the stream gets unrolled, we need to somehow save the yielded elements so that, in the end, it is possible to replicate the processing of these elements from right to left. Therefore, each time a `Yield` is returned by the stream's stepper function, that element needs to be enclosed inside a continuation which encodes what should be performed at that particular point of execution.

```
public class ContinuationListElem<T,S> extends Continuation<S,
     ContinuationListElem<T,S>> {
  public T pendingElem;

  public ContinuationListElem(T elem, Continuation nextCont){
    this.pendingElem = elem;
    this.nextCont = nextCont;
  }
  @Override
  public Continuation execute(S value) {
    Continuation.res = b.apply(pendingElem, value);
    return this.nextCont;
```

When dealing with a list element, a `ContinuationListElem` is used so that the item in question gets saved in `pendingElem`. Note that the execute method, in this case, is responsible for two things:

- applying the parameter function of `fold` to the pending element and the accumulator value;
- returning the next `Continuation` object.

This `Continuation` can be thought of as the `(\a -> cont(f x a))` part in the `foldr'` definition previously presented.

Additionally, we still need another type of `Continuation` in order to have the desired behaviour completely implemented. Another aspect to take into consideration is that, in order to replicate the recursive call in the next iteration of the loop, the self call in `foldr'` needs to be represented as a `Continuation` object too.

```
public class ContinuationListFold<S> extends Continuation<S,
     ContinuationListFold<S>>{

  public ContinuationListFold(Continuation nextCont) {
    this.nextCont = nextCont;
  }
  @Override
  public Continuation execute(S value) {
    return this.nextCont;
```

In this case, the action performed by the execute method is quite simple, as it only returns the next continuation, allowing for the execution flow to take its correct course.

One final aspect to bear in mind is that the chain of continuations ends with a `ContinuationId` which represents the identity function. This final piece is responsible for saving the final accumulator

value to its class variable `res` and not returning a `Continuation`, i.e. `null`.

## 5 STREAM OPTIMIZATION

As previously seen, program fusion generally requires a rule for every possibility of a higher-order function combination. This is not required in *Stream Fusion*. Thus, the refactorings we present are generic.

*Stream Fusion* accomplishes the elimination of intermediate data structures. Although this is the main goal of program fusion, this approach achieves that at the cost of introducing lots of object allocations. In fact, intermediate data structures are replaced by `Step` objects and, depending on the kind of method being executed, complex states like the ones in `appendfs` and `zipfs` need even more objects.

These allocations are responsible for a great amount of overhead. In the Haskell implementation, this situation is handled thanks to several aggressive optimizations included in the Haskell compiler. Therefore, programs are automatically optimized and, in the end, the most efficient solution is obtained (where all the objects mentioned have been eliminated, thus reducing unnecessary allocations).

The Java compiler does not perform any of these optimizations. As a consequence, all the overhead of object allocation, analysis and manipulation is still present in the final executed version.

This section explains how these optimizations can be achieved through Java refactorings [3]. Most Java IDEs provide a great deal of assistance when performing these refactorings [5].

### 5.1 Refactorings

In [1], one of the examples used for demonstration consists of summing the elements of two lists. In their fusion framework, the authors express this as:

```
foldl_s (+) 0 (append_s (stream xs) (stream ys))
```

In Java, this can be translated as:

```
BiFunction<Long, Integer, Long> f = (a, b) -> a+b;
FStream<Integer> xsFs = FStream.fstream(xs);
FStream<Integer> ysFs = FStream.fstream(ys);

Long res = xsFs.appendfs(ysFs).foldl(f, (long) 0);
```

The previous code is how a programmer using the FStream setting would represent this computation. In order to optimize its execution, a set of source code refactorings needs to be applied. To keep the paper concise, the example illustrating the transformations is not included here but it is available from the *FStream* library. The reader is encouraged to check it while reading the summarized description that follows.

The first transformation is to *inline all the stepper functions* that compose the methods `fstream`, `appendfs` and `foldl`. After that, another inlining is applied in which the bodies of the stepper functions of the two `fstream` operations are moved inside the stepper functions corresponding to the `appendfs` operation. Following this, a *case-of-case transformation* is performed where an outer conditional block gets moved and replicated inside the alternatives of a previous inner conditional block. Although this transformation results in the (apparently excessive) duplication of code, it makes it more explicit which path the code from the (now previous) outer conditional block should take. As a result, two out of the three alternatives that compose the block can be completely removed in what can be called a *trivial rewriting* transformation. All of these transformations are repeatedly applied until there are no Step objects left. Also, the majority of Function objects are removed. In case of pipelines that include complex stream states (e.g. appendfs, zipfs) an additional *constructor specialization* transformation may be applied. This aims to eliminate objects that are created only to control the mode of operation (e.g. Left, Right).

## 6 THE GENERAL TEMPLATE

This template consists of a final step in the optimization process which aims to make the code obtained from the refactorings simpler, more readable and more efficient.

### 6.1 The Template's Structure

In order to explain the template's skeleton, we shall consider the following stream pipeline.

```
List<Student> res = fstream(l).filterfs(p).mapfs(f).unfstream();
```

After applying all the optimizations, we obtain a code similar to the one presented below.

The comments highlight the logic behind each instruction.

```
List auxState = l;
boolean over = false;

while (!over) {
  if (auxState.isEmpty()) over = true; //End reached: break out
      of loop
  else {
    // Set sub to be the tail of the current state
    List<Student> sub = auxState.subList(1, auxState.size());

    if (((Predicate) p).test(auxState.get(0))) {
      // auxState.get(0) is the element corresponding to the
      current iteration
      res1.add(f.apply((Student) auxState.get(0)));
      auxState = sub; // Advance one element, i.e. set the state
      to its tail
    } else { auxState = sub; }
```

**Listing 1: Code after optimizations**

If we analyse the code above and try to identify a general form for its representation, we can conclude that the template one whishes to obtain after performing code optimization has the following structure.

```
List auxState = list;
boolean over = false;

while (!over) {
  if (auxState.isEmpty()) over = true;
  else {
    List<T> sub = auxState.subList(1, auxState.size());
    body
  }
```

**Listing 2: Template's structure**

Here, *body* (highlighted inside a red rectangle) consists of the instructions that, when executed, perform the behaviour that the original stream pipeline describes. Therefore, the innermost *if-else* statement in Listing 1 corresponds to the *body* just described.

More precisely, that particular *if-else* statement is the equivalent for the `filter` operation used in the stream version. In a similar

way, the equivalent for the `map` operation is the `f.apply(...)` present inside the *if* statement.

This structure can be converted into a *for loop*.

```
for(... x: list){
    modified body
}
```

**Listing 3: Conversion into for loop**

In Listings 1, 2 and 3 there are some yellow and green highlights that represent the collection (list) over which operations are performed and the element being manipulated in each iteration, respectively.

The collection which the *for loop* iterates over is pretty straightforward to set, as it corresponds to the first value held by the state.

The *while loop* in 1 should perform operations on no more than one element on each iteration. The way we refer to the element in question is through the instruction `auxState.get(0)`. In the *enhanced for loop*, we refer to each element in `list` as `x`. Therefore, each specific occurrence of `auxState.get(0)` gets replaced with `x` (both in the loop's header and in *modified body*). As an iterator is used under the hood for this kind of control flow structure, there is no need to explicitly set what the state should be before the next iteration starts and, as a result, `auxState = sub` instructions do not show up in *modified body*.

Following this idea, the example presented in the beginning can be converted into an equivalent *for loop*.

```
for(Student s: l)
  if (((Predicate) p).test(s)){ res1.add(f.apply(s)); }
```

Going back to the introductory example of the `all` function, if one applies the presented refactorings and templates, an equivalent version using a more efficient *for each* loop is obtained.

```
Boolean value = true;
for(Integer i: xs){ value = value && p.apply(i); }
```

## 7 CONCLUSIONS

In this paper, we expressed functional stream fusion in an object oriented setting by developing a Java library based on the use of lambda expressions. Like Stream Fusion, our embedding does not rely on any specific functional fusion rules - as required by most short-cut fusion techniques - nor in any meta-programming optimizations. Our higher-order functions (map, filter, folds, etc) are modeled directly in Java as operations that work on streams. We also showed the extensability/expressiveness of our techniques by implementing operators, such as foldr and zip, which are not regular Java expressions. The main goal of this paper was to show how Stream Fusion can be expressed in Java. As future work, we plan to study the performance of our implementations in detail. Even though our work does not behave exactly like Stream Fusion, it acts in a very similar way and thus represents a proof of concept showing that these fusion techniques can be equally implemented in an object oriented setting. Currently, most refactorings are provided by Java IDEs and our transformation process is semi-automatic. Now, we plan to develop a plugin to fully automate the process.

## ACKNOWLEDGMENTS

This work is financed by the ERDF European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016718.

## REFERENCES

[1] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/1291151.1291199

[2] João Paulo Fernandes, Alberto Pardo, and João Saraiva. 2007. A Shortcut Fusion Rule for Circular Program Calculation. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. ACM, New York, NY, USA, 95–106. https://doi.org/10.1145/1291201.1291216

[3] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. 2013. LAMBDAFICATOR: From Imperative to Functional Programming Through Automated Refactoring. In *Proc. of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1287–1290.

[5] Robert Fuhrer, Frank Tip, and Adam Kie, un. 2004. Advanced Refactorings in Eclipse. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 8–8. https://doi.org/10.1145/1028664.1028669

[6] Neil Ghani and Patricia Johann. 2008. Short Cut Fusion of Recursive Programs with Computational Effects.. In *Symposium on Trends in Functional Programming (TFP 2008)*.

[7] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

[8] John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-passing Styles. In *Proc. of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 458–471.

[9] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. 1997. Tupling Calculation Eliminates Multiple Data Traversals. In *International Conference on Functional Programming*. 164–175.

[10] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (April 1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

[11] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 285–299. https://doi.org/10.1145/3009837.3009880

[12] Alberto Pardo, João Paulo Fernandes, and João Saraiva. 2016. Multiple Intermediate Structure Deforestation by Shortcut Fusion. *Sci. Comput. Program.* 132, P1 (Dec. 2016), 77–95.

[13] Alberto Pardo, João Paulo Fernandes, and João Saraiva. 2011. Shortcut fusion rules for the derivation of circular and higher-order programs. *Higher-Order and Symbolic Computation* 24, 1 (01 Jun 2011), 115–149.

[14] Francisco José Torres Ribeiro. 2018. *Java stream optimization through program fusion.* Master's thesis. Department of Informatics, University of Minho, Portugal.

[15] Akihiko Takano and Erik Meijer. 1995. Shortcut deforestation in calculational form. In *In Proc. Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 306–313.

[16] Janis Voigtländer. 2008. Semantics and Pragmatics of New Shortcut Fusion Rules. In *FLOPS '08: Proceedings of the 2008 International Symposium on Functional and Logic Programming*. Springer-Verlag, 163–179.

[17] Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (Jan. 1988), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A