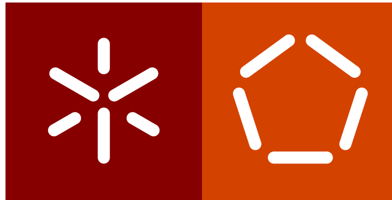


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Francisco José Torres Ribeiro

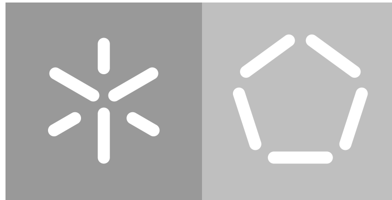
Java *Stream* Optimization Through Program Fusion

September 2018

This work is funded by ERDF - European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the FCT - Foundation for Science and Technology within the project FCOMP-01-0124-FEDER-020484 and grant ref. BI2-2017_PTDC/EEI-ESS/5341/2014_UMINHO.

Cofinanciado por:

UNIÃO EUROPEIA
Fundo Europeu
de Desenvolvimento Regional



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Francisco José Torres Ribeiro

Java *Stream* Optimization Through Program Fusion

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
João Alexandre Baptista Vieira Saraiva
Rui Alexandre Afonso Pereira

September 2018

ACKNOWLEDGEMENTS

Firstly, I would like to thank my family for all the support throughout all these years. They always encouraged and cheered for me, which helped a lot and made it possible for me to be where I am now - completing my Master's degree.

Secondly, I wish to thank both of my supervisors, João Saraiva and Rui Pereira. Their expertise and knowledge helped me overcome the problems I encountered during this work. Additionally, their professionalism and availability to clarify my doubts and to assist with difficulties I encountered along the way contributed heavily to the quality of this thesis.

Furthermore, I would like to also thank Alberto Pardo, who was always looking forward to hearing about my work's latest developments and ready to provide useful and pertinent advice.

To all *Green Software Lab* members, whose comments helped me improve my work.

A special thanks to Bill, my dog, who was always ready to give me a warm welcome every time I got home after a very long week away from him.

Finally, a special thank-you goes to my friends from "Família da Cantina". Their company brought good humour and joy to every meal at the canteen, which was a very important part of this five year journey.

ABSTRACT

Combining different programs or code fragments is a natural way to build larger programs. This allows programmers to better separate a complex problem into simple parts. Furthermore, by writing programs in a modular way, we increase code reusability.

However, these simple parts need to be connected somehow. These connections are done via intermediate structures that communicate results between the different components, harming performance because of the overhead introduced by the allocation and deallocation of multiple structures.

Fusion, a very commonly used technique in functional programming, aims to remove the creation of these unnecessary structures, as they don't take part in the final result.

With the introduction of streams and lambda expressions, Java made its way into a more functional programming style. Yet, these mechanisms lack optimization and the adaptation of fusion techniques used by some compilers for functional languages could benefit the performance of Java streams.

In this thesis, we study how functional fusion can be adapted to Java Streams.

RESUMO

Combinar diferentes programas ou fragmentos de código é uma forma natural de construir programas maiores. Isto permite aos programadores melhor separar um problema complexo em partes simples. Além disso, ao escrever programas de forma modular, estamos a aumentar a reutilização do código.

Contudo, estas partes têm de ser ligadas de alguma maneira. Estas conexões são feitas via estruturas intermédias que comunicam os resultados entre os diferentes componentes, prejudicando a *performance* com o *overhead* introduzido pela alocação e desalocação de várias estruturas.

A fusão, uma técnica muito usada em programação funcional, pretende remover a criação destas estruturas desnecessárias, uma vez que não tomam parte no resultado final.

Com a introdução de *streams* e expressões *lambda*, o Java abriu caminho para um estilo de programação mais funcional. Mesmo assim, estes mecanismos não possuem otimização e a adaptação de técnicas de fusão utilizadas por alguns compiladores de linguagens funcionais poderiam beneficiar a *performance* das *streams* do Java.

Nesta dissertação, é estudado como a fusão em programação funcional pode ser adaptada às *streams* do Java.

CONTENTS

1 INTRODUCTION 1
- 1.1 Context 1
- 1.2 Motivation 1
- 1.3 Objectives 2
- 1.4 Green Software Laboratory 2
- 1.5 Structure of the thesis 3

2 STATE OF THE ART 4
- 2.1 Intermediate structures 4
- 2.2 Deforestation 4
- 2.3 Short-cut fusion 5
- 2.4 Circular program calculation 7
- 2.5 Circular program calculation and short-cut fusion 10
- 2.6 Stream Fusion 12
- 2.7 Hylo System 15

3 FUNCTIONAL STREAM FUSION APPLIED TO JAVA STREAMS 17
- 3.1 Adaptation of the Stream Fusion framework 17
  - 3.1.1 FStream and Step classes 17
  - 3.1.2 FStream methods 19
- 3.2 Overcoming foldr's recursion 37
  - 3.2.1 Creating an intermediate list 38
  - 3.2.2 Foldr as Foldl 38
  - 3.2.3 Using continuations 40
- 3.3 Adaptation to other data structures 44
  - 3.3.1 Binary Tree FStream methods 45

4 STREAM OPTIMISATION 52
- 4.1 Using FStream 52
- 4.2 Inlining all the stepper functions 53
- 4.3 Inlining inside stepper functions 54
- 4.4 Case-of-case transformation 55
- 4.5 Trivial rewriting 57
- 4.6 Repeated application of the rules 58
- 4.7 Constructor specialisation 60
- 4.8 Simplifying the loop 61
- 4.9 Existing IDE refactoring tool 62

5 THE GENERAL TEMPLATE 64
5.1 Considering complex states 64
5.2 Simpler operations 65
5.3 The template's structure 65
5.4 Streams with folds and/or unfolds 67
6 EXPRESSIVENESS 71
6.1 Fibonacci 71
7 PERFORMANCE TESTS 73
7.1 Foldl-Append 73
7.2 Filter-Map 74
7.3 Iterate-Zip 76
7.4 Foldr 77
7.5 Fibonacci 78
7.5.1 Applying transformations 79
7.5.2 Version comparison 80
7.6 Initial experiments 82
7.6.1 Experiment setup 82
7.6.2 Results 84
7.6.3 Discussion 85
8 CONCLUSION/FUTURE WORK 86
8.1 Conclusions 86
8.2 Prospect for future work 88

LIST OF FIGURES

Figure 1 Runtime results for *foldl append* 74
Figure 2 Runtime results for *filter map* 75
Figure 3 Runtime results for *iterate-zip* 77
Figure 4 Runtime results for different *foldr* implementations 78
Figure 5 Runtime results for *Fibonacci* before and after optimisations 80
Figure 6 Runtime results for *recursive*, *recursive fold*, *anamorphism/catamorphism* and *hylomorphism* 81
Figure 7 Runtime results for *iterative* and *tuple* 82
Figure 8 Filter: chained vs merged results 84

LIST OF LISTINGS

2.1 foldr/build rule . . . 6
2.2 pfold/buildp for leaf trees . . . 11
5.1 Code after optimisations . . . 65
5.2 Template's structure . . . 66
5.3 Conversion into for loop . . . 66
5.4 optimised unfoldr based stream . . . 68

1

INTRODUCTION

1.1 CONTEXT

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

Recently, Java adopted lambda expressions as a mechanism to manipulate its collections, the so called streams. These lambda expressions are very used by the functional programming community and allow writing traversals on complex data structures in a concise way.

However, the execution of these recursion patterns has several efficiency problems, either by doing more traversals than necessary, or by creating intermediate data structures. In the context of functional programming, a lot of work has been developed for an efficient execution of these mechanisms. Unfortunately, none of this work was incorporated in Java and, because of that, stream execution is not efficient, yet!

In recent times, the growing concern with energy waste has steered the focus of software optimization not only to performance issues, like execution time and memory consumption, but to the energy consumption side as well. As such, it is expected that the adaptation of several fusion mechanisms (well known to the functional community) will allow its application on Java streams, making it possible to contribute to a greater efficiency of the programs both in terms of energy consumed and execution time.

1.2 MOTIVATION

With the introduction of *streams* and lambda expressions, Java allowed programmers to write their code in a more concise and readable way. Moreover, the Stream class allows the possibility to compose operations, such as *maps* and *filters*, making it possible to express long, and sometimes complex, sequences of instructions with little effort. However, if these mechanisms are not tuned appropriately, they may lead to efficiency problems. As such,

programs are affected by certain problems such as the introduction of overhead, multiple traversals and allocation of needless objects.

Modifying the program's code in order to overcome these issues has the drawback of compromising its readability. Furthermore, a more efficient implementation may not necessarily be the most natural solution to a problem, leading to increased difficulty during development and maintenance.

In essence, programmers wish to write programs in the style they are most familiar with, not necessarily the most efficient one, and have them perform the best way possible. They want the best of both worlds.

Therefore, there's a need for techniques that automatically perform these kinds of optimisations automatically. That is where this thesis tries to step in, providing a way for Java programmers to overcome the efficiency problems that arise as a consequence of them writing their programs in powerful ways that ease their understandability.

1.3 Objectives

In this thesis, the following objectives are to be achieved:

- Adapt fusion and deforestation techniques to Java streams
- Develop a structured template to assist in the systematic application of fusion and deforestation techniques to streams
- Validate this template with programs using streams
- Make a detailed study of the gains obtained with the optimisation, namely in terms of execution time.

1.4 Green Software Laboratory

This thesis is being developed under a research grant within the *Green Software Lab (GSL)* project, which aims to analyse and reduce energy consumption in software systems. *GSL* is funded by *Fundação para a Ciência e a Tecnologia (FCT)*.

In the context of previous work done under the *GSL* project (which is not presented in this thesis), I was a co-author of the following publications:

- Towards a Green Ranking for Programming Languages [2] in the *21st Brazilian Symposium on Programming Languages (SBLP)*, which was awarded the Best Paper Award, leading to the submission of an extended version to *SCP 2018*.

- Energy Efficiency across Programming Languages: How does energy, time, and memory relate? [15] in the *10th International Conference on Software Language Engineering (SLE)*
- Energyware Analysis [3] in the *7th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA)*

1.5 STRUCTURE OF THE THESIS

This document is structured as follows.

Chapter 2 presents the state of the art, where some important contributions to *fusion* and *deforestation* are presented so as to place ourselves in the current stage of development of these techniques.

Chapter 3 is the one in which the code produced in order to adapt *Stream Fusion* to an object-oriented setting is discussed.

Chapter 4 explains how some of *GHC's* optimisations were mimicked as code refactorings.

Chapter 5 presents the template one whishes to obtain after applying the transformations described in the previous chapter.

Chapter 6 presents some examples in which the adapted setting provides a higher degree of functional expressiveness to Java.

Chapter 7 compares the impact of the different optimisation steps by presenting the obtained results from the performance tests that were carried out. Additionally, it presents some experiments performed in the early stages of this work in order to assess the current behaviour of Java Streams.

Chapter 8 closes this document by stating the attained conclusions and providing some insights to where future work could be directed.

2

STATE OF THE ART

2.1 INTERMEDIATE STRUCTURES

Building programs through function composition has many advantages as it makes it easier to write programs in a clear and modular way. However, this style of programming is subject to some *runtime* penalties [4]. More precisely, each function composing the program needs to communicate its result to the next function. In functional programming, a paradigm which is very fond of this kind of programming approach, this is often accomplished by creating intermediate lists that connect the different functions assembling the program. Therefore, with strict evaluation, a lot of intermediate structures are allocated along the way which do not take part in the final result [19].

As Wadler [19] states, the problem with the memory usage of these structures can be overcome with lazy evaluation. This way, because elements are generated as they are needed, there is no requirement for loading the entirety of the intermediate lists. However, each list element still has to be allocated, checked and de-allocated.

2.2 DEFORESTATION

To deal with the problem arising from the allocation of intermediate lists, a program transformation technique called *deforestation* is used. This method allows for the elimination of intermediate structures which are created and consumed soon afterward.

One of the first deforestation algorithms was presented by Wadler and, although it removed intermediate data structures, it had some disadvantages, as Gill et al. state.

The major drawback of this kind of approach to the elimination of intermediate structures is the restriction imposed on the algorithm inputs. In his paper, Wadler [19] presents what he calls a *treeless* form for defining functions which do not use any internal intermediate structures. The algorithm developed transforms a program composed by functions defined in *treeless* form into a single function, also defined in *treeless* form. As one can see, this is where one of the main disadvantages of this technique is evident. By limiting its application

to functions defined in a restrictive form, the algorithm has a restricted range of inputs to operate on.

This places boundaries on the style of programming allowed to programmers, compromising code readability and conciseness.

A technique allowing the elimination of intermediate data structures, and thus creating a more efficient version, without sacrificing code clarity was needed.

Saraiva and Swierstra [17] studied the elimination of intermediate structures in the context of attribute grammars.

2.3 SHORT-CUT FUSION

In Gill et al. [7], a transformation technique to create more efficient versions of programs through the elimination of intermediate lists is presented. The core idea behind this deforestation technique are the algebraic transformations performed on some functions.

With these algebraic transformations, the authors show it is possible to standardise the way lists are consumed and produced. Furthermore, this algorithm allows every program as input.

In Haskell, one could define the well known list data type as:

```
data List a = Nil | Cons a (List a)
```

`foldr` is a function which behaviour consists of processing a list with an operator and returning the value it constructed along the way (accumulated in an initial value).

This systematic consumption of a list can be thought of as replacing every occurrence of `Cons` with the provided operator and the `Nil` instance with the initial value.

Therefore, many functions that consume lists in a constant way like the one just described can be expressed in terms of `foldr`. That is because this higher-order function encloses that kind of systematic consumption of a list.

Example implementations resorting to `foldr` of pre-defined functions can be seen in the same paper by Gill et al. [7].

However, this standardisation of list consumption is not enough to achieve the desired program transformation, as the following example demonstrates.

Supposing a composition of functions like:

```
sum (map f ls)
```

where `map` applies function `f` to each element of `ls` and `sum` performs the addition of every element in the list.

One could modify this program and have:

```
foldr (+) 0 (foldr ((:).f) [] ls)
```

where `foldr` is a higher-order function which consumes a recursive data structure (in this case, a list) by applying a given combining function in a systematic way to all the constituent parts, building a return value in the end.

But there isn't a rule that simplifies occurrences of *foldr/foldr*. A workaround for this, could be rewriting these kinds of programs in a more specific way, and have the above example transformed in:

```
foldr ((+).f) 0 ls
```

The problem with this approach is that it is not very general. More precisely, it is very difficult to be sure we have sufficient rules. When another combination of functions is encountered, a new rule would need to be written so that that particular case would get simplified.

In the example used to illustrate this, the `foldr` on the outside had no way to know how the `foldr` on the inside was producing its result list. As such, we also need a way to standardise list production.

The abstraction described for list consumption consists in the replacement of every `cons` with a function and the `nil` at the end with a given value. And `foldr` encapsulates this behaviour by receiving a function `f` and an initial value `acc`.

Therefore, if list production is abstracted in terms of `cons` and `nil`, it is possible to obtain `foldr`'s effect if this list-producing abstraction is applied to `f` and `acc`.

As such, a function `build` can be defined like:

```
build g = g (:) []
```

Following the line of thought just described, we come up with the *foldr/build* rule, which can be expressed as:

```
foldr f acc (build g) = g f acc
```

Listing 2.1: foldr/build rule

As an example, one can consider the `upto` function which, given two numbers, produces a list that starts from the first one and continues until the second one.
In a very straightforward way, one could define this function as:

```
upto x y = if x>y then []
           else x : upto (x+1) y
```

But, as stated before, we can try to abstract the production of the list in terms of `cons` and `nil`, and thus getting the following definition:

```
upto' x y = \ cons nil -> if x>y then nil
                          else cons x (upto' (x+1) y cons nil)
```

Now, the function `upto` would be written like:

```
upto x y = build (upto' x y)
```

Deforestation is now possible if the list is produced using `build` and consumed using `foldr`:

```
mul (upto x y) = foldr (*) 1 ( build (upto' x y))
               = upto' x y (*) 1
```

Applying the *foldr/build* rule presented in 2.1 (key elements highlighted inside red rectangles) allows us to obtain a reduced form of the function `mul`, where no intermediate list is produced, which confirms the effect of deforestation.

2.4 CIRCULAR PROGRAM CALCULATION

Algorithms that perform multiple traversals on the same data structure can be expressed as a single traversal function through a technique called circular program calculation.
This kind of approach, first explored by Bird [1], highlights the importance of the lazy evaluation mechanism in functional languages like Haskell. In fact, defining circular programs in this way only works because of lazy evaluation. A circular definition has the consequence of creating a function call containing an argument that is, simultaneously, a result of that same call. Under a strict evaluation mechanism, this can be a problem as an

infinite cycle is created because values are demanded before they can be calculated, leading to non-termination.

On the other hand, lazy evaluation allows for the computation of such circular structures. With this strategy, the right evaluation order of the expression is determined at runtime. More specifically, only the elements of the expression to be computed that are necessary to continue are expanded.

Although circular programs avoid unnecessary multiple traversals, they are not necessarily more efficient than their more straightforward counterparts [6] and are even more difficult to write. In fact, even more experienced programmers find it hard to understand programs written in such a way. In his paper, Bird proposes deriving these circular programs from their less efficient (in terms of number of traversals), but more natural, equivalent solutions.

The example he uses is the function `repmin`, which has become a traditional example for being simple and a good assistant for the explanation of this particular technique.

The problem at hand is going to be the replacement of every leaf value in a tree with the original minimum value of the tree.

First of all, we must define a datatype for the tree. After that, we need a function `replace` and a function `tmin` to swap the tree's leaves for a given value and to calculate the minimum value of a tree, respectively.

With that, we can easily come up with a natural way of expressing the problem, which is implemented by the function `transform`.

```
data LeafTree = Leaf Int
              | Fork (LeafTree, LeafTree)

tmin :: LeafTree → Int
tmin (Leaf n) = n
tmin (Fork (l, r)) = min (tmin l) (tmin r)

replace :: (LeafTree, Int) → LeafTree
replace (Leaf _, m) = Leaf m
replace (Fork (l, r), m) = Fork (replace (l, m),
                                 replace (r, m))

transform :: LeafTree → LeafTree
transform t = replace (t, tmin t)
```

After having a straightforward solution to the problem, one can start applying Bird's proposed technique.

The first step consists of tupling. The functions `tmin` and `replace` both have a similar recursive pattern and operate on the same data structure. Therefore, a function `repmin` can be created by combining the results from the two previous functions in a tuple.

```
repmin (t, m) = (replace (t, m), tmin t)
```

Furthermore, a recursive definition of this function can be created, in which two cases need to be considered:

```
repmin (Leaf n, m)
    = (replace (Leaf n, m), tmin (Leaf n))
    = (Leaf m, n)

repmin (Fork (l, r), m)
    = (replace (Fork (l, r), m), tmin (Fork (l, r)))
    = (Fork (replace (l, m), replace (r, m)) , min (tmin l) (tmin r))
    = (Fork (l', r'), min n1 n2)
            where (l', n1) = repmin (l, m)
                  (r', n2) = repmin (r, m)
```

The final step is where circular programming is used in order to put together the two elements forming the result of `repmin`.

Highlighted inside blue rectangles is the presence of circularity; `m` is being used simultaneously as an argument and a result of the same call.

```
transform :: LeafTree → LeafTree
transform t = nt
    where (nt,m) = repmin (t,m)
```

This method for deriving circular programs presents, however, a drawback.

Although it allows the derivation of a circular program from a more natural and intuitive equivalent, removing the burden of having to come up with such a complicated implementation and creating a circular alternative which makes less traversals on the data structure, this technique does not guarantee termination. In order to illustrate this, in his paper, Bird even presents a case where this happens.

2.5 CIRCULAR PROGRAM CALCULATION AND SHORT-CUT FUSION

A different technique than the one explained previously is presented by Fernandes et al. [5]. This approach also has in mind the derivation of circular programs from more natural ones, but through a different process.

The method in question is applied to programs consisting of the composition of functions $f \circ g$, where:

- g is a producer of type: $g :: i \rightarrow (b, v)$
- f is a consumer of type: $f :: (b, v) \rightarrow r$

These programs have a particular circumstance compared to the ones we talked about in 2.3. In addition to an intermediate structure, the functions composing the program need to communicate through an additional parameter.

The *repmin* problem constitutes a very good case to explain strategies concerning circular programming. As such, it will again be used as an example.

The strategy presented in this section is applied to programs defined in a slightly different way than the definition used for the *repmin* problem in 2.4. Thus, in order to make that program suitable for the application of this method, the original, and more natural, solution needs to be adapted so that the function types coincide with the ones of f and g presented above.

Function `replace` maintains its original definition, as its type is already equivalent to $f :: (b, v) \rightarrow r$.

However, function `tmin` needs to be modified:

```
transform :: LeafTree → LeafTree
transform t = replace ∘ tmint $ t

tmint :: LeafTree → (LeafTree, Int)
tmint (Leaf n) = (Leaf n, n)
tmint (Fork (l, r)) = (Fork (l', r'), min n1 n2)
        where (l', n1) = tmint l
              (r', n2) = tmint r

replace :: (LeafTree, Int) → LeafTree
replace (Leaf, m) = Leaf m
replace (Fork (l, r), m) = Fork (replace (l, m),
                                 replace (r, m))
```

This way, the producer and the consumer are defined in the way we need and the main function is expressed as the composition of those two functions, just like intended.

As the title of the section suggests, this method applies short-cut fusion in order to derive circular programs.

In the paper by Fernandes et al. [5], a generic rule for such fusion is provided. For the example in question, this rule can be expressed as:

```
pfold T (h1, h2) ∘ buildp T g $ c = v
      where (v, z) = g (k1, k2) c
             k1 n = h1 (n, z)
             k2 (l, r) = h2 ((l, r), z)
```

Listing 2.2: pfold/buildp for leaf trees

Therefore, in order to meet the conditions necessary to apply this rule, the consumer (`replace`) needs to be written in terms of a *pfold* and the producer (`tmint`) needs to be written in terms of a *buildp*.

```
replace = pfold (Leaf ∘ π2, Fork ∘ π1)

tmint = buildp g
    where g (leaf, fork) (Leaf n) = (leaf n, n)
          g (leaf, fork) (Fork (l, r)) = let (l', n1) = g (leaf, fork) l
                                             (r', n2) = g (leaf, fork) r
                                         in (fork (l', r'), min n1 n2)
```

Now that the functions are expressed in the necessary form, we can apply rule 2.2, thus obtaining:

```
transform t = nt
    where (nt, m) = g (k1, k2) t
           k1 = Leaf m
           k2 (l, r) = Fork (l, r)
```

For clarity, if we replace the above function definition with the actual code being executed:

```
transform t = nt
    where (nt, m) = repm t
          repm (Leaf n) = (Leaf m, n)
          repm (Fork (l, r)) = let (l', n1) = repm l
                                   (r', n2) = repm r
                               in (Fork (l', r'), min n1 n2)
```

The circularity introduced in this final version of the program is a consequence of merging the construction of the result tree and the minimum value.

With this version, the production of an intermediate data structure (a tree in this case) has been eliminated and the original tree is traversed only once. This technique eliminates both unnecessary intermediate structures and multiple traversals.

One of the main points to retain about the difference between the method by Fernandes et al. [5] and the method by Bird [1] is the safe introduction of circularity. That is, the lazy engine can safely schedule the computations so that termination is guaranteed, a consequence of the way in which circularity is introduced.

This assumption comes from a property between `tmint` and `repm`, which lies in the fact that both are able to compute the minimum value of a tree without depending on the respective tree.

```
π2 ∘ tmint = π2 ∘ repm
```

This property holds in general and, as such, does not apply to this particular case only. This way, the method can be applied to a wide range of programs.

Circular programs and short-cut fusion are also studied in other works, such as [11], [12], [13], [9] and [14].

2.6 Stream Fusion

The work by Coutts et al. [4] in *Stream Fusion* consists of an automatic deforestation system that takes a different approach compared to more traditional short-cut fusion systems.

The approach taken by Gill et al. [7] with the *foldr/build* rule is to fuse functions that work directly over the original structure of the data, that is, lists.

In *Stream Fusion*, the operations over the original list structure are transformed in order to, instead, work over the co-structure of the list.

As Coutts et al. 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:

```
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 and 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.

In order to convert list structures to streams and vice-versa, two functions are needed.

```
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 `next0` which is non-recursive and yields each element of the stream as it unfolds;
- a state, which consists of the list itself.

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

Implementing functions to perform operations over streams is quite simple. The function intended has to define the particular stepper function for the stream it is going to return as a result. Considering the simple and well known `map` example operating on lists, one would define its stream counterpart as:

```
map s :: (a → b) → Stream a → Stream b
map s f (Stream next0 s0) = Stream next s0
    where
        next s = case next0 s of
              Done → Done
              Skip s' → Skip s'
              Yield x s' → Yield (f x) s'
```

What `map` does here is define a stepper function that applies the function given as a parameter of `map` to every yielded element of the stream.

A very simple but important case where one can see the effect of the stream fusion approach is the function `filter`. Its implementation allows us to observe the true impact of this technique.

```
filter s :: (a → Bool) → Stream a → Stream a
filter s p (Stream next0 s0) = Stream next s0
    where
        next s = case next0 s of
            Done → Done
            Skip s' → Skip s'
            Yield x s' | p x → Yield x s'
                       | otherwise → Skip s'
```

The only way that the function `filter` is non-recursive is because of `Skip`. This constructor, when put in place of the elements that should be removed from the stream, allows us to avoid the recursion otherwise necessary to process every stream element in order to find out which ones satisfy the given predicate.

More precisely, in the last line of the above implementation, `Skip` is introduced whenever an element does not pass the predicate's test.

This way, code can be better optimised by general purpose compiler optimisations.

The method documented so far has a very curious and important implication.

To understand it, we should first be aware that the Glasgow Haskell Compiler allows us to write rules that will then be used while compiling our programs. These "custom rules" can be expressed through pragmas, which are instructions that can be given to the compiler.

Some algebraic transformations can be expressed through these pragmas. For example:

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

This algebraic rule expresses that a composition of maps is equivalent to the map of the composition of the two functions. This rule allows for the generated code to be more efficient.

However, there is a multitude of possible function combinations and, as a consequence, one could never be certain of the number of rules necessary to cover all cases.

This is a point where the work by Coutts et al. plays an important role. As presented earlier, when writing different stream combinators (like `map` and `filter`), the outcome of each stepper function that is defined depends on the outcome of the previous stream's stepper function.

This way, whenever a stepper function of a stream is called, every stepper function of the streams preceding the current one is going to be executed.

Therefore, functions are fused without the need to explicitly state the rules performing those transformations.

2.7 HYLO SYSTEM

Program calculation is what is behind the techniques described to transform programs into more efficient versions. These techniques are based on many existing transformation laws. However, these rules only allow us to work with programs by hand, therefore leaving the application of program transformations necessary to obtain more efficient versions to the programmer, and not to the computer. Fusion systems are, as a consequence, not automatic.

Algorithms need to be developed that construct programs based on those transformation laws. This is what the HYLO system by Onoue et al. [10] aims to be: a fusion system applying these transformations in a more universal and regular way than existing ones.

First of all, we need to understand that there are two possible approaches to fusion: search-based fusion and calculational fusion.

The first one, search-based fusion, unfolds recursive definitions of functions to find suitable places inside those expressions to perform folding operations. But to achieve this, this kind of method needs to keep track of the function calls so it can control the unfolding, in order to avoid an infinite process. As this introduces a great overhead, fusion cannot be practically implemented this way.

The work by Onoue et al. [10] focuses on the second kind of fusion, calculational fusion, which has been the object of a lot of investigation over the years.

This approach explores the recursive structure of each component of the program in order to apply fusion through existing transformation laws.

However, most of the proposed techniques for fusion have the slight inconvenient of forcing the programmer to express the functions in terms of the necessary recursive structure,

so that the different transformations can be applied. This is impractical, as it leads the programmer away from more potentially readable and natural implementations.

With this in mind, when briefly explaining their approach, the authors of the HYLO system start by stating that the majority of recursive functions can be expressed in terms of a very specific recursive form: hylomorphism. In order to rewrite the program's recursive components in terms of hylomorphisms, the authors developed an algorithm to derive such general recursive structures from the recursive definitions of the program.

Following that, schemes for data production and consumption need to be captured so that the *Acid Rain Theorem* can be applied to hylomorphisms, in order to fuse them.

The final step consists of inlining the resulting hylomorphism into a normal recursive definition, in which the intermediate structures have been eliminated.

All in all, the HYLO system allows programs to be written without the concern of expressing them in terms of specific and more generic recursive structures, as these are derived by an automatic algorithm. Thus, fusion laws can still be applied, leading to more efficient programs without sacrificing so much code readability and without forcing programmers to express functions under certain recursive patterns. This system was incorporated into the Haskell compiler.

3

FUNCTIONAL STREAM FUSION APPLIED TO JAVA STREAMS

3.1 ADAPTATION OF THE STREAM FUSION FRAMEWORK

The work by Coutts et al. [4] presents an automatic deforestation system in Haskell, called *stream fusion*, that is achieved by fusing different operations on lists. The paper describes the Haskell code created for that purpose, so the first step of the implementation for this thesis consists on the mapping of that Haskell code to an equivalent Java code.

3.1.1 *FStream and Step classes*

In the paper, the Stream and Step datatypes are defined as:

```
data Stream a = ∃ s. Stream (s → Step a s) s
data Step a s = Done
              | Yield a s
              | Skip s
```

As one can 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; //stepper function: (s -> Step a s)
  public Object state; // the stream's 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, there is 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;
    }
  }
```

3.1.2 *FStream methods*

After creating the necessary functional datatypes in the Java implementation, there has to be a way to generate an `FStream` object from a known state. To not complicate our presentation, we start by considering the state to be a list datatype.

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

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

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> l){
    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, l);
}
```

In order to create a list back from a stream, the `unstream` function repeatedly calls the stepper function of the stream, unfolding it.

```
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 most important part of this function is the `unfold`. In the equivalent Java method that was created, this unfolding behaviour was implemented in the form of a `while` loop, since a tail recursive function, which is one where its last action is the call to itself, can be more efficiently expressed by a loop. The stream's stepper function is repeatedly called inside the loop. Depending on the result of that method call, 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's state.

```
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 [8]. They are a powerful mechanism that allow programmers to define what the computations *are* instead of programming the steps that compose the computation.

Maps and *filters* are two well known higher order functions and this approach has implementations for those two functions over streams. `map f xs` applies input function `f` to each element of `xs` and `filter p xs` returns a list with all the elements of `xs` that fulfill the condition stated by `p`.

The Haskell implementation for the *map* higher order function over streams is:

```
maps :: (a → b) → Stream a → Stream b
maps f (Stream next0 s0) = Stream next s0
    where
        next s = case next0 s of
            Done → Done
            Skip s' → Skip s'
            Yield x s' → Yield (f x) s'
```

As expected, 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.

In the case 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);
}
```

In order to supplement the explanation, the following example illustrates how the $\texttt{map}_s$ function would be used in a Haskell and in a Java scenario. In this case, we add `1` to each element of a stream.

```
xs = [1,2,3,4,5]
res = unstream (maps (+1) (stream xs))
-- res = [2,3,4,5,6]
```

```
ArrayList<Integer> xs = new ArrayList<>(Arrays.asList(1,2,3,4,5));

ArrayList<Integer> res = FStream.fstream(xs)
                                .mapfs(x -> x + 1)
                                .unfstream();
// res = [2,3,4,5,6]
```

The `filter` higher order function is the one where the big advantage of having a non-recursive stepper function can be observed. 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. This way, *GHC* can better optimise 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 `filterfs` method 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 of 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);
}
```

The following example demonstrates how we can retain the elements greater than 2 from a stream by resorting to the higher-order function $\texttt{filter}_s$. This is illustrated using both the Haskell and the Java alternative.

```
xs = [1,2,3,4,5]
res = unstream . filter_s (>2) . stream $ xs
-- res = [3,4,5]
```

```
ArrayList<Integer> xs = new ArrayList<>(Arrays.asList(1,2,3,4,5));

ArrayList<Integer> res = FStream.fstream(xs)
                                .filterfs(x -> x > 2)
                                .unfstream();
// res = [3,4,5]
```

Concatenating two lists is a common operation. Haskell makes use of the `++` operator in order to express this action. The *Stream Fusion* library implements this same functionality for *streams* through the `append`$_s$ function. The `append`$_s$ function makes use of the `Either` datatype in order to concatenate two different streams. This is crucial to maintain the control flow of the operation.

Thus, `Left` and `Right` make it possible for the stepper function to work on two distinct modes, that is, holding the state of the first stream or the state of the second stream.

When the end of the first stream's state is reached, the stepper function simply switches its mode from `Left` to `Right`, allowing for an easy exchange between the two different states.

```
append_s :: Stream a → Stream a → Stream a
append_s (Stream next_a s_a0) (Stream next_b s_b0) =
  Stream next (Left s_a0)
      where
          next (Left s_a) =
              case next_a s_a of
                  Done → Skip (Right s_b0)
                  Skip s'_a → Skip (Left s'_a)
                  Yield x s'_a → Yield x (Left s'_a)

          next (Right s_b) =
              case next_b s_b of
                  Done → Done
                  Skip s'_b → Skip (Right s'_b)
                  Yield x s'_b → Yield x (Right s'_b)
```

In Java, there is not an equivalent class emulating Haskell's `Either` datatype. However, this is a trivial thing to implement. First, an abstract class for `Either` is created.

```
public abstract class Either<L,R> {
      L left;
      R right;
  }
```

After that, classes `Left` and `Right` extend it, as follows:

```
public class Left<T> extends Either{

      public Left(T e){
          this.left = e;
      }

      public T fromLeft(){
          return (T) this.left;
      }
  }
```

```
public class Right<T> extends Either{

      public Right(T e){
          this.right = e;
      }

      public T fromRight(){
          return (T) this.right;
      }
  }
```

In the Haskell code above, we can see that $\texttt{append}_s$'s stepper function makes use of *pattern matching* to assess in which mode it is operating on.

In Java, we perform this check in a similar way to what has been presented so far when checking whether `Step` objects are of type `Done`, `Skip` or `Yield`. More precisely, depending on whether the stream's state is an `instanceof Left` or an `instanceof Right`, the actions corresponding to `next (Left` s_a`)` or `next (Right` s_b`)` are executed.

```
public FStream<T> appendfs(FStream<T> streamB){
    Function<Object, Step> nextAppend = x -> {
        if(x instanceof Left){
            Step aux = this.stepper.apply(((Left) x).fromLeft());

            if(aux instanceof Done){
                return new Skip<Either>(new Right(streamB.state));
            }
            else if(aux instanceof Skip){
                return new Skip<Either>(new Left(aux.state));
            }
            else if(aux instanceof Yield){
                return new Yield<T, Either>((T) aux.elem, new Left(aux.state
                    ));
            }
        }
        else if(x instanceof Right){
            Step aux = streamB.stepper.apply(((Right) x).fromRight());

            if(aux instanceof Done){
                return new Done();
            }
            else if(aux instanceof Skip){
                return new Skip<Either>(new Right(aux.state));
            }
            else if(aux instanceof Yield){
                return new Yield<T, Either>((T) aux.elem, new Right(aux.
                    state));
            }
        }

        return null;
    };

    return new FStream<T>(nextAppend, new Left(this.state));
  }
```

The following example shows a way in which this function could be used to concatenate the elements from two different streams, both in Haskell and Java.

```
xs = [1,2,3,4,5]
ys = [6,7,8,9,10]
res = unstream (append_s (stream xs) (stream ys))
-- res = [1,2,3,4,5,6,7,8,9,10]
```

```
ArrayList<Integer> xs = new ArrayList<>(Arrays.asList(1,2,3,4,5));
ArrayList<Integer> ys = new ArrayList<>(Arrays.asList(6,7,8,9,10));

ArrayList<Integer> res = FStream.fstream(xs)
                                .appendfs(FStream.fstream(ys))
                                .unfstream();
// res = [1,2,3,4,5,6,7,8,9,10]
```

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 $\texttt{zip}_s$ 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)*.

```
public class Triple<T,S> {
      T stateA;
      S stateB;
      Optional elem;

      public Triple(T stateA, S stateB, Optional elem) {
          this.stateA = stateA;
          this.stateB = stateB;
          this.elem = elem;
      }
  }
```

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.

```
public class Pair<T,S> {
      T x;
      S y;

      public Pair(T x, S y){
          this.x = x;
          this.y = y;
      }
  }
```

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()));
  }
```

An example that demonstrates the use of the $\texttt{zip}_s$ function is the enumeration of elements in a list ys. In other words, we produce a list of pairs in which the first element of each tuple is the corresponding index of the second element in the original list ys.

Note that list xs, which holds the indexes, is larger than ys. That is to show that when the two lists differ in length, the bigger one gets its remaining elements discarded, i.e. they are not part of the final result.

```
xs = [0,1,2,3,4,5,6,7]
ys = [10, 20, 30, 40, 50]
res = unstream (zips (stream xs) (stream ys))
-- res = [(0,10),(1,20),(2,30),(3,40),(4,50)]
```

```
ArrayList<Integer> xs = new ArrayList<>(Arrays.asList(0,1,2,3,4,5,6,7));
ArrayList<Integer> ys = new ArrayList<>(Arrays.asList(10,20,30,40,50));

ArrayList<Pair<Integer,Integer>> res = FStream.fstream(xs)
                                .zipfs(FStream.fstream(ys))
                                .unfstream();
// res = [(0,10),(1,20),(2,30),(3,40),(4,50)]
```

Some higher-order functions have to deal with nested data structures. For example, the function concatMap from the standard Haskell list library applies its input function to each element of its input list, similarly to what the map function does. However, in this case, this input function produces a list, instead of a single element. As such, concatMap concatenates every list resulting from the application of its input function. If it didn't, we would obtain a list of nested lists. $\texttt{concatMap}_s$ is an example of a function on nested streams. The function f it receives is applied to each one of the stream's elements (similarly to $\texttt{map}_s$). However, this function has a particular aspect: it does not produce a single value, but a stream of values.

The goal is to produce a stream of single elements and not a stream of streams. Therefore, we need to have some special concerns when implementing a function like this.

The function operates on two different modes, much like what has been presented above in other functions.

The two modes are: applying function f to the elements of the outer stream, generating another stream, and extracting the elements from each generated inner stream. To differentiate between these modes, the datatype Maybe is used.

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

When the stepper function `nextConcatMap` is unfolding the outer stream, this `Optional` value is set to empty.

When applying the function to one of its elements, the generated stream needs to be unfolded before continuing to process the outer stream.

Therefore, the `Optional` value is set to hold the new inner stream, making the stepper function switch modes. As a consequence, it is then going to unfold that inner stream, switching back to the outer one when it reaches the end.

```java
public <S> FStream<S> concatMap(Function<T, FStream<S>> f){
      Function<Object, Step> nextConcatMap = x -> {
          Optional opAux = (Optional) ((Pair) x).getY();

          if(!(opAux.isPresent())){
              Step aux = this.stepper.apply(((Pair) x).getX());

              if(aux instanceof Done){
                  return new Done();
              }
              else if(aux instanceof Skip){
                  return new Skip(new Pair<>(aux.state, Optional.empty()));
              }
              else if(aux instanceof Yield){
                  return new Skip(new Pair<>(aux.state, Optional.of(f.apply
                      ((T) aux.elem))));
              }
          }
          else{
              FStream fAux = (FStream) opAux.get();
              Step aux = (Step) fAux.getStepper().apply(fAux.getState());

              if(aux instanceof Done){
                  return new Skip(new Pair(((Pair) x).getX(), Optional.empty
                      ()));
              }
              else if(aux instanceof Skip){
                  return new Skip(new Pair(((Pair) x).getX(), Optional.of(
                      new FStream<>(fAux.getStepper(), aux.state))));
              }
              else if(aux instanceof Yield){
                  return new Yield(aux.elem, new Pair(((Pair) x).getX(),
                      Optional.of(new FStream<>(fAux.getStepper(), aux.state
                      ))));
              }
          }

          return null;
      };

      return new FStream<>(nextConcatMap, new Pair(this.state, Optional.
          empty()));
  }
```

For a demonstration of `concatMap`, let's assume the existence of a function `upTo` that receives a single integer `x` as an argument and that generates a stream representing numbers from 1 to `x`.

```
xs = [1,2,3]
res = unstream (concatMap_s upTo (stream xs))
-- res = [1,1,2,1,2,3]
```

```
ArrayList<Integer> xs = new ArrayList<>(Arrays.asList(1,2,3));

ArrayList<Integer> res = FStream.fstream(xs)
                                .concatMap(upTo)
                                .unfstream();
// res = [1,1,2,1,2,3]
```

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

As seen before with `unstream`, these terminal operations are implemented in a recursive way.

```
foldr_s :: (a → b → b) → b → Stream a → b
foldr_s f z (Stream next s0) = go s0
    where
        go s = case next s of
                Done → z
                Skip s' → go s'
                Yield x s' → f x (go s')
```

```
foldl_s :: (b → a → b) → b → Stream a → b
foldl_s f z (Stream next s0) = go z s0
    where
        go z s = case next s of
                    Done → z
                    Skip s' → go z s'
                    Yield x s' → go (f z x) s'
```

In the Java implementation, recursion is converted to loops whenever possible. That is the case with `unstream` and `foldl`. However, `foldr` is a particular case because it is not *tail*

recursive. As a consequence, it cannot be expressed in terms of a loop in the same manner the other two functions are.

One could implement `foldr` by making it traverse the list in a reverse way, that is, from right to left. However, because *Stream Fusion* retrieves elements as they are needed, the list that `foldr` is supposed to process still does not exist. One cannot invert the order of the input list at this stage either because the preceding operations have not yet been executed and that would create an incorrect result.

One possible way would be to force the creation of the list before applying the final `foldr` operation. This would have a negative impact on performance, as this intermediate structure is not required to produce the final result. Moreover, this is the kind of situations that fusion aims to avoid.

In section 3.2, we discuss in great detail how to express `foldr` in Java using loops instead of recursion.

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

`foldr` behaves in an identical way to `foldl`, except that the elements are processed in the opposite way.

As discussed earlier, instead of a loop, there are recursive calls.

```
public <S> S foldr(BiFunction<T,S,S> f, S value){
    return goFoldr(f, value, this.stepper, this.state);
}


public static <S, R> S goFoldr(BiFunction<R, S, S> f, S value, Function<
    Object, Step> stepper, Object state) {

    Step step = stepper.apply(state);

    if (step instanceof Done) {
        return value;
    } else if (step instanceof Skip) {
        return goFoldr(f, value, stepper, step.state);
    } else if (step instanceof Yield) {
        return f.apply((R) step.elem, goFoldr(f, value, stepper, step.
            state));
    }

    return null;
}
```

One is not only limited to use the `fstream` method as the first operation in a stream pipeline.

Another alternative for retrieving elements is the `unfoldr` method.

```
unfoldr :: (b → Maybe (a, b)) → b → Stream a
unfoldr f s0 = Stream next s0
    where
        next s = case f s of
            Nothing      → Done
            Just (w, s') → Yield w s'
```

Function `f` acts as a builder that generates elements for the stream and is called each time the stepper function is executed.

It returns either `Nothing`, if it is done generating elements, or `Just` containing a tuple in which the first value is the generated element and the second one is the value to be considered when the next call occurs.

```
public static <T,S> FStream<T> unfoldr(
  Function<S, Optional<Pair<T,S>>> builder,
  S seed){
    Function<Object, Step> nextUnfoldr = x -> {
        Optional<Pair<T, S>> aux = builder.apply((S) x);

        if(!aux.isPresent()){
            return new Done();
        }
        else{
            return new Yield<>(aux.get().getX(), aux.get().getY());
        }
    };

    return new FStream<T>(nextUnfoldr, seed);
}
```

The equivalent `FStream` method is represented above. Its mapping to Java is pretty straightforward. The `Optional` class is used as a replacement for `Maybe` and the `Pair` class is used to emulate Haskell tuples.

3.2 Overcoming foldr's recursion

In a functional setting, a recursive implementation for `foldr` is, undoubtedly, the most intuitive one. Furthermore, when programming in a functional context, recursive implementations arise very naturally.

Nevertheless, consecutive recursive calls rapidly fill up the stack and stack overflow errors may happen.

As stream pipelines may very well describe operations to be performed in collections with a large number of elements, repeated consecutive calls are bound to happen in terminal operations that are not implemented in the form of a loop, like `foldr`'s case. As such, applying a pipeline of operations terminating with `foldr` on a large list will most likely cause the program to raise a stack overflow error.

Indeed, the main responsible for these errors is not recursion itself, but the nature of recursion. In [4], other terminal operations, like `unstream` and `foldl`, are expressed in terms of recursion too. However, these functions are *tail recursive*. As there is no need for the call stack to increase in size, functions of this type can be easily transformed into their equivalent loops.

The original implementation of `foldr` is not the case. As such, we need a slightly more elaborate process if we want to have it written as a loop. As a consequence of doing that, we will overcome the inherent stack overflow errors that the non-tail recursive implementation presents.

3.2.1 *Creating an intermediate list*

One way to process the list's elements from right to left is to traverse the list from the end to its beginning.

However, because we are using streams, that approach is not so straightforward as it may seem because there is no actual list. With streams, elements are returned one at a time, as they are needed. Furthermore, they depend on the operations that compose the pipeline.

As such, if we are to traverse the list in reverse order, we need to force its creation before `foldr`.

```
public <S> S foldrLoop(BiFunction<T,S,S> f, S value){
    List<T> l = this.unfstream();

    // From size-1 to 0 (right to left)
    for(int i = l.size()-1; i>=0; i--){
        T t = l.get(i);
        value = f.apply(t, value);
    }

    return value;
}
```

Highlighted in blue[1] is the use of `unfstream` which creates the list that results from applying the operations that precede `foldr`. Only then can it be traversed from right to left, emulating the desired behaviour.

3.2.2 *Foldr as Foldl*

Contrary to `foldr`, `foldl` has a straightforward implementation in the form of a loop (its original definition is tail recursive).

An alternative way to tackle the current problem is to express `foldr` in terms of a `foldl`. Although this may look tricky at first sight, the strategy used to achieve this is quite simple.

1 We assume that colours are available in the electronic version of this document.

At a higher degree of abstraction, *fold* simply consists of a recursive pattern. Instead of applying such a pattern directly to our data structure, we can use it to build a function along the way. In the end, a "big" function will have been constructed. This function will be composed of several function applications referencing the list's elements in the correct order. When applied to the accumulator's initial value, the order in which the function was composed will make its execution yield a result that resembles what would have been obtained if the elements were processed from right to left.

```
foldrAsFoldl f z lxs = foldl (\g x -> (\a -> g(f x a))) id lxs z
```

```
public <S> S foldrAsFoldl(BiFunction<T,S,S> f, S value){
    BiFunction<Function<S,S>, T, Function<S,S>> reducer =
        (g, x) -> (a -> g.apply(f.apply(x,a)));

    Function<S,S> finalAcc = this.foldl(reducer, Function.identity());
    return finalAcc.apply(value);
}
```

The first highlight in blue is a *lambda* expression representing the reducer function that will be passed to `foldl`. It receives two parameters, one of them being a function (`g`) and the other one an individual element from the stream (`x`). What is so particular about this reducer is that it does not return a value, but a function. The function consists of the application of input function `g` to the value returned by function `f` when applied to stream element `x` and the accumulator `a`. This returned function is going to be `g` in the next execution. One can now understand that this mechanism is the one responsible for composing all the operations in the appropriate order.

Also highlighted in blue is the call to `foldl` that, besides the reducer, takes the *identity* function as a parameter which, in this case, behaves as the accumulating value we are used to see associated with *folds*.

However, this solution does not completely overcome stack overflow errors but, instead, delays them. Because the function built along the way consists of consecutive nested function calls, the stack's size increases and we are bound to encounter the same problems. The main advantage of this implementation is that the function size only grows when `Yield` elements are encountered. As such, pipelines containing methods which produce `Skip` objects (for example, `filter` operations) will be able to operate on larger collections than the original `foldr` implementation.

3.2.3 *Using continuations*

There is another possible implementation for `foldr` that tries to avoid its inherent recursion problems. This solution resorts to *Continuation Passing Style* [18], [16] in order to have the code explicitly state what should be executed next. By having the control flow programmed this way, we avoid filling up the stack because function calls are encapsulated inside objects that represent the *continuations*. Furthermore, it eases the process of implementing *foldr* in the form of a loop, as it allows us to define it as a *tail-recursive* function.

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

This definition for *foldr* will help in understanding the reason why *continuations* were encoded the way they were.

The first thing to do is to build some classes that will allow us to encode the desired behaviour. Moreover, we need to combine this style of programming with the *FStream setting*.

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 starts with the *identity* function.

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

    public ContinuationId(){
    }

    @Override
    public Continuation execute(S value) {
        res = value;
        return null;
    }
}
```

This *identity* function is represented by the `ContinuationId` class. As a result of repeatedly adding Continuations one after the other, this initial *identity* Continuation is going to be pushed to the very end of the chain, thus terminating the execution by saving the final accumulator value to its class variable `res` and not returning any type of `Continuation`.

Now that the implementation of *Continuation Passing style* was explained, we can take a look at the mapping of the *tail-recursive* version of `foldr'` to a loop.

```
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;
}
```

Similarly to what happens in other `stream` operations, the loop begins by applying the stepper function to the current state.

As we progress through the stream, by retrieving elements from it, a chain of continuations is constructed. The part of the code responsible for the creation of this chain is highlighted in green. When a `Yield` object is present, a new `Continuation` is instantiated and placed at the beginning of the chain.

When the `stream` reaches its end, with the detection of `Done`, the head of the continuation chain gets its `execute` method called, unwinding the execution of all the other `Continuation`s.

3.3 ADAPTATION TO OTHER DATA STRUCTURES

Defining streams for other data structures is something that can be addressed by the *Stream Fusion* approach. In fact, in the end of their paper, Coutts et al. [4] briefly cover what could be a definition for a stream co-structure on binary trees with information present in the leaves and the interior nodes.

In this section, the elements that need to be added in order to adapt *Stream Fusion* to binary trees only containing information in the leaves will be presented. Although this kind of binary trees differs from the ones mentioned in the paper, it still shows that streams can be generalised to other data structures.

The binary trees considered in this section are expressed by the following data type:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

The `Stream` data type is identical to the one already presented. The only thing that needs some adjustments is the `Step` data type for which the constructors `LeafBT` and `BranchBT` were introduced.

```
data Stream a = ∃s. Stream (s → Step a s) s
data Step a s = LeafBT a | BranchBT s s | Skip s
```

Each of these data constructors was mapped to a corresponding Java class.

```
public class LeafBT<T> extends Step{

    public LeafBT(T elem) {
        this.elem = elem;
    }
}
```

```
public class BranchBT<S> extends Step{
    public S state1;
    public S state2;

    public BranchBT(S state1, S state2) {
        this.state1 = state1;
        this.state2 = state2;
    }
}
```

3.3.1 *Binary Tree FStream methods*

Similarly to what was previously shown for `FStream` over lists, it is also possible to write *stream style* code for methods operating on streams for binary trees.

In order to create an `FStream` capable of providing elements as if we were dealing with a binary tree, we can use the method `unfoldrBT`.

A Haskell definition for an *unfold* that generates binary trees looks like the following (note that this is not Stream code):

```
unfoldT :: (b → a + b x b) → (b → Tree a)
unfoldT g x = case g x of
   Left a → Leaf a
   Right (c, d) → Branch (unfoldT g c) (unfoldT g d)
```

Here, one can see that the function returns either `Leaf` or `Branch` depending on whether it has reached an end node or it is supposed to go further down and generate more elements for the tree. This is, of course, going to depend on what the initial seed is (first value for `x`) and what the builder function `g` does with each value.

By looking at the above definition we can deduce that the programmer should define its builder function in order to have it return `Either`.

We still need to make minor adjustments to this function in order to have it fit our `FStream` setting. Firstly, we need to define a *stepper* function, as every `FStream` needs one. Secondly, our stepper function needs to return `Step` elements. `LeafBT` and `BranchBT` defined earlier already fit the current structure of `Leaf` and `Branch` quite nicely, so we do not need to change this part. What's left is to fit the `case g x of...` and its body (with a small change) inside a stepper function. By doing this, we will end with the following method.

```
public static <T,S> FStream<T> unfoldrBT(
  Function<S, Either<T, Pair<S,S>>> builder,
  S seed){
    Function<Object, Step> nextUnfoldrBT = x -> {
        Either<T, Pair<S,S>> aux = builder.apply((S) x);

        if(aux instanceof Left){
            return new LeafBT<>(((Left) aux).fromLeft());
        }
        else if(aux instanceof Right){
            Pair p = (Pair) ((Right) aux).fromRight();
            return new BranchBT(p.getX(), p.getY());
        }

        return null;
    };

    return new FStream<T>(nextUnfoldrBT, seed);
}
```

As it is the usual behaviour for streams, the stepper function is going to be repeatedly called when the terminal operation of a pipeline gets executed. As such, when the builder function returns an element of type `Right`, `unfoldrBT` does not get recursively called like in the above Haskell code (as recursion is actually "prohibited" in stream style code for intermediate operations) but instead returns a `Step` of type `BranchBT` which will hold two separate states.

Conversely, it is also possible to define the `fold` pattern for binary trees in the `FStream` setting.

In order to do this, we base ourselves again in some Haskell code. This time, we'll take a look at `foldr` for binary trees.

```
fold_T :: (b → b → b) → (a → b) → (Tree a → b)
fold_T l (Leaf a) = l a
fold_T b l (Branch s t) = b (fold_T b l s) (fold_T b l t)
```

$\texttt{fold}_T$ receives two functions as parameters. Depending on whether the current element is a `Leaf` or a `Branch`, the corresponding function is applied.

After making some adjustments, namely calling the stream's stepper function and covering the case when `Skip` is returned, the following implementation is obtained.

```
public <S> S foldBT(BiFunction<S,S,S> b, Function<T,S> l) {
    RecursiveLambda<Function<Object,S>> go = new RecursiveLambda<>();
    go.function = x -> {
        S res = null;
        Step step = this.stepper.apply(x);

        if (step instanceof LeafBT) {
            res = l.apply((T) step.elem);
        } else if (step instanceof Skip) {
            res = go.function.apply(step.state);
        } else if (step instanceof BranchBT) {
            res = b.apply(go.function.apply(((BranchBT) step).state1), go.
                function.apply(((BranchBT) step).state2));
        }

        return res;
    };

    return go.function.apply(this.state);
}
```

Note that, unlike what we wish when dealing with terminal operations, this implementation is not based on a loop but, instead, makes use of recursion. More precisely, it is not tail-recursive, which holds up our ability to express it in the form of a loop.

Furthermore, binary trees propose a new challenge. Earlier, when dealing with streams that worked over lists, performing the *fold* operation from left to right helped us implement the solution as a loop in a really straightforward way. But now, with the introduction of binary trees, we cannot think of it that way anymore. `foldBT` ”destroys” the structure from the leaves and upwards, whereas a left-fold would consume it from the root and downwards. This latter approach does not make much sense when dealing with binary trees.

Therefore, we are left with the option of finding an equivalent tail-recursive implementation. This is not a new problem and some ways to overcome it have been discussed earlier.

For this particular case, the stack was ”moved” into the heap. More precisely, a structure of type `Stack` was used in order to regulate the control flow.

This way, we can emulate the order in which the recursive calls would be executed in the previous recursive implementation and have our code represented as a loop.

```
public <S> S foldlBT(BiFunction<S,S,S> b, Function<T,S> l) {
    S res = null;
    boolean over = false;
    Stack<Object> states = new Stack<>();
    states.push(this.state);
    Optional<S> opAux = Optional.empty();

    while(!over){
        if(states.empty()){
            res = opAux.get();
            over = true;
        }

        else{
            Step step = this.stepper.apply(states.pop());

            if(step instanceof LeafBT){
                if(!opAux.isPresent()){
                    opAux = Optional.of(l.apply((T) step.elem));
                }
                else{
                    opAux = Optional.of(b.apply(opAux.get(), l.apply((T)
                        step.elem)));
                }
            }
            else if(step instanceof BranchBT){
                states.push(((BranchBT) step).state2);
                states.push(((BranchBT) step).state1);
            }
        }
    }

    return res;
}
```

The implementation also makes use of an `Optional` object, which begins by having the empty value (Haskell's `Nothing`), to assess when it is the first time dealing with a `LeafBT`. If it happens to be the case, `opAux` is assigned to carry an actual value (Haskell's `Just`). The next time a `LeafBT` is encountered, a value will be present in `opAux`, and function `b` can be applied.

Highlighted in blue are the `push` operations performed on the stack, which correspond to the self calls in the recursive version.

Expressing the binary tree *fold* as a loop is also possible through the use of *Continuation Passing Style* (as seen with `foldr` for streams operating over lists).

Similarly to what was done in that situation, specialised continuations need to be created in order to address the specific cases of the control flow.

As we did previously, we will first take a look at the *tail-recursive* version of the *fold* operation for binary trees in Haskell.

```
foldTTail _ l (Leaf a) cont = cont (l a)
foldTTail b l (Branch s t) cont =
    foldTTail b l s (\a1 -> foldTTail b l t (\a2 -> cont (b a1 a2)))
```

As `BranchBT` objects hold two states, the continuations containing calls to *fold* need to have a reference to one of the states so that when they are executed, they can change the variable `globalState` in order to steer the course of execution in the right direction.

Furthermore, the continuation in question also needs to indicate its `nextCont` what the current pending element is, so that it knows which element to apply function `b` to.

In the end, it returns the next continuation, as usual.

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

    public Object state;


    public ContinuationFold(Object state, Continuation nextCont) {
        this.state = state;
        this.nextCont = nextCont;
    }

    @Override
    public Continuation execute(S value) {
        globalState = this.state;
        ((ContinuationBranchOp) this.nextCont).pendingElem = value;
        return this.nextCont;
    }
}
```

By the definition above, it can be seen that `(\a2 -> cont (b a1 a2))` should also correspond to some type of `Continuation` that applies the function to both sides of the branch.

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

    public ContinuationBranchOp(Continuation nextCont){
        this.nextCont = nextCont;
    }

    @Override
    public Continuation execute(S value) {
        return nextCont.execute((S) b.apply(pendingElem, value));
    }
}
```

As we previously saw, when `foldr` for streams over lists was presented, a chain of `Continuation` objects needs to be constructed throughout the stream unrolling process.

Again, the initial element of this chain is `ContinuationId`, representing the *identity* function.

```
public <S> S foldBTTailRec(BiFunction<S,S,S> b, Function<T,S> l) {
      Continuation.b = b;
      Continuation cont = new ContinuationId();
      boolean over = false;
      Continuation.globalState = this.state;

      while(!over){
          Step step = this.stepper.apply(Continuation.globalState);

          if(step instanceof LeafBT){
              cont = cont.execute(l.apply((T) step.elem));

              if(cont == null){
                  over = true;
              }
          }
          else if(step instanceof BranchBT){
              Continuation.globalState = ((BranchBT) step).state1;

              Continuation<S,ContinuationBranchOp<S>> nextCont =
                  new ContinuationBranchOp<>(cont);

              cont = new ContinuationFold(((BranchBT) step).state2, nextCont
                  );
          }
      }

      return (S) Continuation.res;
  }
```

Although the implementation above differs a little from the majority of stream operations, one can still find some similarities.

As usual, the stream's stepper function is applied at the beginning of every iteration.

`Continuation`s are added every time a step of type `BranchBT` is returned. Here, it is important to store the second state (`state2`) the branch refers to. That way, the `Continuation` object in question has a reference that will allow it to point the execution in the right direction.

Every time a step of type `LeafBT` is detected, it means that one of the tree's bottoms has been reached and the top most `Continuation` in the chain gets executed.

4

STREAM OPTIMISATION

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.

Intermediate data structures are replaced by `Step` objects and, depending on the kind of method being executed, complex states like the ones in `appendfs`, `zipfs` and `concatMap` need even more objects in order to maintain the correct mode of operation.

These allocations are going to be responsible for a great amount of overhead. In the Haskell implementation, this situation is overcome thanks to several optimisation techniques included in *GHC*. Therefore, programs are automatically transformed 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 optimisations. As a consequence, all the overhead of object allocation, analysis and manipulation is still present in the final executed version.

This chapter will explain how these optimisations can be achieved through *source code refactoring*.

4.1 USING FSTREAM

In Coutts et al. [4], 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);
```

This Java code will be the starting point for all the optimisation steps presented from this point forward.

4.2 INLINING ALL THE STEPPER FUNCTIONS

The first step is to inline all the stepper functions that compose the methods `fstream`, `appendfs` and `foldl`.

The code presented below shows that the creation of the respective streams from the two lists `xs` and `ys` corresponds to the inlining of the stepper functions `nextStream` and `nextStream1`.

Similarly, `appendfs` gets inlined in the form of its respective stepper function `nextAppend`. Depending on whether the state has the type `Left` or `Right`, we can see that the appropriate stepper function is called: `nextStream` or `nextStream1`.

The body of these functions is omitted fully or in part because it is equivalent to what has been already documented in 3.

We can then see that the seed value gets set to 0 and the initial state to `Left(xs)`, indicating that list `xs` is going to be processed first.

Finally, `foldl` gets inlined as a loop that repeatedly calls the `nextAppend` stepper function.

```
Function<Object, Step> nextStream = x -> { ... };

Function<Object, Step> nextStream1 = x -> { ... };

Function<Object, Step> nextAppend = x -> {
    if(x instanceof Left){
        Step aux = nextStream.apply(((Left) x).fromLeft());
        ...
    }
    else if(x instanceof Right){
        Step aux = nextStream1.apply(((Right) x).fromRight());
        ...
    }
};

Long value = (long) 0;
Object auxState = new Left(xs);
boolean over = false;

while (!over) {
    Step step = nextAppend.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, (Integer) step.elem);
    }
}

Long res = value;
```

4.3 INLINING INSIDE STEPPER FUNCTIONS

In this step, `nextStream` and `nextStream1` get inlined inside `nextAppend` (highlighted in blue), which is the stepper function responsible for calling the first two.

For demonstration purposes, only the `Left` mode is presented below. However, an identical transformation takes place for the `Right` branch.

It is now clearer that, depending on the outcome of `nextStream` (now inlined), `aux` will be either `Done` or `Yield` which will determine the path taken in the last conditional block. Therefore, this refactoring created a nested conditional.

```
Function<Object, Step> nextAppend = x -> {
    if(x instanceof Left){
        Step aux = ((Function<Object, Step>) x1 -> {
            List aux1 = (List) x1;

            if (aux1.isEmpty()) {
                return new Done();
            } else {
                List<Integer> sub = aux1.subList(1, aux1.size());
                return new Yield<Integer, List<Integer>>((Integer) aux1.
                    get(0), sub);
            }
        }).apply(((Left) x).fromLeft());

        if(aux instanceof Done){
            return new Skip<Either>(new Right(ys));
        }
        else if(aux instanceof Skip){
            return new Skip<Either>(new Left(aux.state));
        }
        else if(aux instanceof Yield){
            return new Yield<Integer, Either>((Integer) aux.elem, new Left
                (aux.state));
        }
    }
    ...
}
```

4.4 CASE-OF-CASE TRANSFORMATION

In the previous section, there were two main conditional blocks: an inner one that checks if the list is empty, and an outer one that checks the `Step` type.

By applying the *case-of-case transformation* we push the outer block inside each one of the alternatives of the inner block, thus obtaining the following code (outer case insertion highlighted in blue).

```
Function<Object, Step> nextAppend = x -> {
    if(x instanceof Left){
        Step aux = ((Function<Object, Step>) x1 -> {
            List aux1 = (List) x1;

            if (aux1.isEmpty()) {
                Step innerAux = new Done();

                if(innerAux instanceof Done){
                    return new Skip<Either>(new Right(ys));
                }
                else if(innerAux instanceof Skip){
                    return new Skip<Either>(new Left(innerAux.state));
                }
                else if(innerAux instanceof Yield){
                    return new Yield<Integer, Either>((Integer) innerAux.
                        elem, new Left(innerAux.state));
                }
            } else {
                List<Integer> sub = aux1.subList(1, aux1.size());
                Step innerAux = new Yield<Integer, List<Integer>>((Integer
                    ) aux1.get(0), sub);

                if(innerAux instanceof Done){
                    return new Skip<Either>(new Right(ys));
                }
                else if(innerAux instanceof Skip){
                    return new Skip<Either>(new Left(innerAux.state));
                }
                else if(innerAux instanceof Yield){
                    return new Yield<Integer, Either>((Integer) innerAux.
                        elem, new Left(innerAux.state));
                }
            }

            return null;
        }).apply(((Left) x).fromLeft());
        ...
```

Although this particular transformation duplicates code significantly, it makes it easier to apply another transformation where the code gets rewritten in a more concise way.

4.5 TRIVIAL REWRITING

By applying the *case-of-case transformation* previously, we made it more explicit that it is possible to determine the path taken by the outer conditional block.

In the previous code, if `aux1.isEmpty()` evaluates to *true*, then `innerAux` will be a `Done` object. Consequently, it becomes evident that `innerAux instanceof Done` will be *true*. Thus, the only alternative being executed will be the body of that particular *if statement*.

Therefore, all the other code can be removed. Following this line of thought, we can transform the portion of code we have been using so far into the following.

```
Function<Object, Step> nextAppend = x -> {
    if(x instanceof Left){
        Step aux = ((Function<Object, Step>) x1 -> {
            List aux1 = (List) x1;

            if (aux1.isEmpty()) {
                return new Skip<Either>(new Right(ys));
            } else {
                List<Integer> sub = aux1.subList(1, aux1.size());
                return new Yield<Integer, Either>((Integer) aux1.get(0),
                   new Left(sub));
            }
        }).apply(((Left) x).fromLeft());

        return aux;
    }
    ...
```

As stated previously, for conciseness reasons, the transformations applied have been demonstrated in detail only for the `Left` branch.

If we follow the reasoning documented in the previous sections, we get the following code for the `Right` branch, which resembles the code listed prior for the `Left` mode.

```
...
else if(x instanceof Right){
    Step aux = ((Function<Object, Step>) x1 -> {
        List aux1 = (List) x1;

        if (aux1.isEmpty()) {
            return new Done();
        } else {
            List<Integer> sub = aux1.subList(1, aux1.size());
            return new Yield<Integer, Either>((Integer) aux1.get(0), new
                Right(sub));
        }
    }).apply(((Right) x).fromRight());

    return aux;
}
```

So far, the previous three transformations (*inlining*, *case-of-case* and *trivial rewriting*) have avoided the allocation and check of one `Step` object for each iteration.

4.6 Repeated application of the rules

These transformations can be applied again. As such, we inline the current form of `nextAppend` into the loop, apply the *case-of-case* rule and finally rewrite the code.

```
final Long[] value = {(long) 0};
final Object[] auxState = {new Left(xs)};
final boolean[] over = {false};

while (!over[0]) {
    Step step = ((Function<Object, Step>) x -> {
        if (x instanceof Left) {
            Step aux = ((Function<Object, Step>) x1 -> {
                List aux1 = (List) x1;

                if (aux1.isEmpty()) {
                    auxState[0] = new Right(ys);
                } else {
                    List<Integer> sub = aux1.subList(1, aux1.size());
                    auxState[0] = new Left(sub);
                    value[0] = f.apply(value[0], (Integer) aux1.get(0));
                }

                return null;
            }).apply(((Left) x).fromLeft());

            return aux;
        } else if (x instanceof Right) {
            Step aux = ((Function<Object, Step>) x1 -> {
                List aux1 = (List) x1;

                if (aux1.isEmpty()) {
                    over[0] = true;
                } else {
                    List<Integer> sub = aux1.subList(1, aux1.size());
                    auxState[0] = new Right(sub);
                    value[0] = f.apply(value[0], (Integer) aux1.get(0));
                }

                return null;
            }).apply(((Right) x).fromRight());

            return aux;
        }

        return null;
    }).apply(auxState[0]);
}

Long res = value[0];
```

If we compare the current state of the program with the first presented version, we can see that some variables (`value`, `auxState` and `over`) have been converted to one-element arrays.

The reason behind this modification are the requirements imposed on the variables handled inside the scope of *lambda expressions*. These expressions can only access variables of the enclosing block that are final. However, the variables cannot simply be converted to *final* because, in addition to being accessed, they also need to be modified.

Thus, for all purpose, by transforming the variables in question to final one-element arrays, we are not assigning a new value to them, but only to one of its features. The array behaves as a wrapper around the values we wish to work with.

Every occurrence of `Step` objects has been removed by now. Some references to this class are still present in the return values of the `Function` objects but they can be considered leftovers of the refactoring. If we inspect the code being executed, we verify that objects of this type are never instantiated nor manipulated. In fact, the only thing the `Function`s do is accessing and modifying the outside variables.

4.7 CONSTRUCTOR SPECIALISATION

Allocation of intermediate `Step` objects has been eliminated by now.

However, some functions also require the allocation of additional objects in order to maintain the stream's state. This is the case with `appendfs` since it resorts to `Either` to control its mode of operation.

Therefore, similarly to what happened before with `Step`, an `Either` object is instantiated on every iteration.

If we take a look at the *if/else statement* in the previous section where we check whether the function is operating on the `Left` or `Right` mode and consider each of those branches to be a specialised version of the loop, we can remove those conditional statements and break the loop into two.

Now, each mode of operation has its own dedicated loop, thus eliminating the need to repeatedly check the instance type of the current state. As such, updating the state can be addressed in a straightforward way and it gets the list assigned directly.

Note that splitting the loop into two does not imply executing more iterations. The first loop only iterates over the first list and, when finished, switches the state to the second list, breaks out and the second loop continues from that point forward.

```
final Long[] value = {(long) 0};
final Object[] auxState = {xs};
final boolean[] over = {false};

while (!over[0]) {
    Step aux = ((Function<Object, Step>) x1 -> {
        List aux1 = (List) x1;

        if (aux1.isEmpty()) {
            auxState[0] = ys;
            over[0] = true;
        } else {
            List<Integer> sub = aux1.subList(1, aux1.size());
            auxState[0] = sub;
            value[0] = f.apply(value[0], (Integer) aux1.get(0));
        }

        return null;
    }).apply(auxState[0]);
}

over[0] = false;
while (!over[0]){
    Step aux = ((Function<Object, Step>) x1 -> {
        ...
    }).apply(auxState[0]);
}

Long res = value[0];
```

4.8 Simplifying the loop

The `subList` method used allows the creation of a partial view over the list we want our program to work on.

Although the creation of this view is relatively cheap, it is still not the most efficient way to iterate over a collection, at least in Java.

Iterating is expressed in a more efficient way through the use of *for-each loops*.

There is no need to be constructing a view of the target list every time we want to look at the next element.

Therefore, the iteration mechanism being used in the implementation so far can be converted to the following.

```
long res = 0L;
  for (Integer i : xs) {
      res = f.apply(res, i);
  }

  for (Integer i : ys) {
      res = f.apply(res, i);
  }
```

Furthermore, the function in question is a very basic one: addition.

Using a `Function` object to encapsulate this kind of behaviour is, in this particular case, subject to some unnecessary overhead.

If we translate the execution of such function, `f.apply(res, i)`, to a direct use of the `+` operator, we get the following and more efficient version of the program.

```
long res = 0L;
  for (Integer x : xs) {
      long l = x;
      res += l;
  }
  long sum = 0L;
  for (Integer i : ys) {
      long l = i;
      sum += l;
  }
  res += sum;
```

4.9 EXISTING IDE REFACTORING TOOL

The previous sections presented a way to refactor an existing program expressed as a chain of different stream operations into an equivalent version using loops which achieved better performance.

Nowadays, IDE's contain several tools that allow programmers to refactor their code in a multitude of ways.

Some of these development environments offer Java Stream API conversion to *for loops* as one of the possible refactors.

As an example, one could obtain the final optimised version of our program by using the included refactoring tool in *IntelliJ IDEA*.

However, in this particular example, the IDE does not seem to detect that this conversion can take place due to the stream concatenation.

For every other example tested that only used one stream, the IDE was able to translate code written with the Java Stream API to the equivalent *for loop* version.

5

THE GENERAL TEMPLATE

This chapter describes the structure of the general template that is obtained after the optimisation steps described in 4 have been performed.

The following sections describe considerations that are important to have in mind when thinking about this process.

5.1 CONSIDERING COMPLEX STATES

The example used for the demonstration in 4 belongs to a set of stream pipelines which has a particular characteristic.

Examples of this kind manipulate what is referred to in *Stream Fusion* as *complex stream states*.

These *non-trivial state types* are necessary in order to guarantee the correct control flow of the operations in question, such as *append* and *zip*.

Different types represent different modes to operate on. Therefore, the corresponding stepper functions need to check the type of the current state they are manipulating. Depending on the result of that inspection, the code corresponding to that mode of operation is executed.

This mechanism requires the creation of objects that represent those different types of complex states. In the example used in 4, `Left` and `Right` object are instantiated in order to achieve the desired effect.

As a result, when optimising the code, it is necessary to remove these allocations, as they are not necessary for the execution. That is precisely what the *constructor specialisation* step achieves. We only obtain the general template after this.

5.2 SIMPLER OPERATIONS

Functions like *filter* and *map* do not manipulate the kind of states mentioned in the previous section. As such, when a stream pipeline is only composed of operations of this nature, the code optimisation process does not require *constructor specialisation*.

As a result, one can obtain the general template in a more straightforward way.

5.3 THE TEMPLATE'S STRUCTURE

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

```
Predicate<Student> p = s -> s.grade >= 95;
Function<Student, Student> f =
    s -> new Student(s.name, Math.round((double) s.grade / 10));


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

After applying all the optimisations, 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 of the list reached: break out of the 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 5.1: Code after optimisations

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 optimisation 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 5.2: 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 5.1 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 5.3: Conversion into for loop

In 5.1, 5.2 and 5.3 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 5.1 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 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 line of thought, 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));
    }
}
```

5.4 Streams with folds and/or unfolds

When the method used to generate an `FStream` object is an `unfold`, and not `fstream`, then the template obtained in the end slightly differs from the one described previously.

The same goes for the cases when the terminal operation of the stream pipeline is a `fold` and not an `unfstream`.

In order to better illustrate the situation, let's consider another example. This time, we consider the well known *factorial* operation implemented in the form of an `unfold` followed by a `fold`.

```
Function<Integer, Optional<Pair<Integer,Integer>>> f = x -> {
    if (x > 0) {
        return Optional.of(new Pair<>(x, x - 1));
    } else {
        return Optional.empty();
    }
};

Integer res = FStream.unfoldr(f, 5).foldl((x, y) -> x * y, 1);
```

`Function f` represents the function which `unfoldr` uses in order to generate the stream's elements. This function gets executed each time the higher order function `unfoldr` needs to return a new element. As such, the `Pair` used inside the builder function holds both the element to be retrieved and the value it needs to consider during its next call.

In this particular example, `x` gets decremented each time and `f` stops returning elements when zero is reached.

After applying the transformations, the following is obtained.

```
Function<Integer, Optional<Pair<Integer,Integer>>> f = x -> {
    if (x > 0) {
        return Optional.of(new Pair<>(x, x - 1));
    } else {
        return Optional.empty();
    }
};

Integer value = 1;
Object auxState = 5;
boolean over = false;

while (!over) {
    Optional<Pair<Integer, Integer>> aux = f.apply((Integer) auxState);

    if (!aux.isPresent()) {
        over = true;
    } else {
        auxState = aux.get().getY();
        value = ((BiFunction<Integer, Integer, Integer>) (x, y) -> x * y).
            apply(value, aux.get().getX());
    }
}

Integer res = value;
```

Listing 5.4: optimised unfoldr based stream

The above program can be converted in an equivalent one using a *for-loop*, presented below. Colours were used in order to highlight where the elements of one version fit in the other one.

```
Integer value = 1;
for(Integer x = 5; x > 0; x = x-1){
    value = ((BiFunction<Integer, Integer, Integer>) (a, b) -> a * b).
        apply(value, x);
}
```

Similarly to what was done in 5.3, we wish to find a way to generalise this transformation to programs of this kind. As such, we try to find a template that conforms to the structure

of such programs. If we examine the code in 5.4 we can locate the most important elements (highlighted in colours). The builder function `f` passed to `unfoldr` has two components:

- blue: the condition that sets whether the function keeps returning elements or not;
- yellow: the operation that sets the value for the next call.

Before the loop starts, there are two other details highlighted:

- red: the accumulator value;
- green: the seed state, i.e the value that is passed when calling `unfoldr`'s builder function for the first time.

After that, the *while-loop* takes over the execution. It encapsulates consecutive calls to the builder function in order to continuously retrieve elements. The outer *if-else* statement controls whether the execution continues or not. If the builder did not return any element, we break out of the loop. Otherwise, the body (highlighted inside a red rectangle), corresponding to all the operations that compose the stream pipeline, gets executed.

```
Function<T, Optional<Pair<T,S>>> f = x -> {
    if (condition_to_continue) {
        return Optional.of(new Pair<>(x, advance_operation));
    } else {
        return Optional.empty();
    }
};

S value = accumulator;
Object auxState = seed_state;
boolean over = false;

while (!over) {
    Optional<Pair<T, S>> aux = f.apply((T) auxState);

    if (!aux.isPresent()) {
        over = true;
    } else {
        body
    }
}

Integer res = value;
```

Based on this structure, it is possible to derive an equivalent control flow making use of a *for-loop*. The highlighted elements are repositioned in order to fill the appropriate spots. Moreover, the body gets modified:

- statements responsible for advancing the state (`auxState = aux.getY().get()`) are eliminated because the loop's iterator working under the hood takes care of that;
- references to `aux.get().getX()` (corresponding to each iteration's variable) are replaced by `x`.

```
S value = accumulator;
for(T x = seed_state; condition_to_continue; advance_operation){
    modified body
}
```

6

EXPRESSIVENESS

The FStream setting allows for a degree of expressiveness that Java Streams do not currently support.

In this chapter, some examples of programs that showcase this expressive power are presented.

6.1 FIBONACCI

Naively, a Haskell implementation stands as follows.

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

We could think of the sequence of calls that a program calling this function would generate and generalise a representation over binary trees.

```
fib_T :: Integer → Tree Integer
fib_T 0 = Leaf 0
fib_T 1 = Leaf 1
fib_T n = Branch (fib_T (n - 1)) (fib_T (n - 2))
```

The above function returns a binary tree (Tree datatype).

A program that calculates Fibonacci numbers could therefore be written as the composition of an *unfold* followed by a *fold*[1].

1 Indeed, the HYLO system converts the original fib implementation into a fold∘unfold.

```
fib'T :: Integer → Integer
fib'T = foldT (+) id ∘ unfoldT g
    where
        g 0 = Left 0
        g 1 = Left 1
        g n = Right (n - 1, n - 2)
```

Equivalently, one could write this program in Java using the *FStream* setting.

```
// Builder function passed to unfoldrBT
Function<Integer, Either<Integer, Pair<Integer,Integer>>> g = x -> {
    if (x == 0){
        return new Left<>(0);
    }
    else if(x == 1){
        return new Left<>(1);
    }
    else{
        return new Right<>(new Pair<>(x-1, x-2));
    }
};

Integer res = unfoldrBT(g, NUMBER)
              .foldlBT((x,y) -> x + y, Function.identity()));
```

7

PERFORMANCE TESTS

In order to assess the effect that each transformation has on the program's performance, each version obtained through the application of the those rules was tested in terms of *runtime* performance.

In addition, the equivalent implementation using Java Streams was also measured.

Although the program took less time to complete with the increasing number of optimisation rules that were applied, in order to show the values in a more concise way, only the results for the original Java Streams implementation, FStreams implementation, the final optimised loop obtained from FStreams (after the transformations) and the final *for-each loop* are going to be presented.

7.1 FOLDL-APPEND

In *Stream Fusion*, one of the examples used by Coutts et al. [4] to demonstrate the transformations performed by the *GHC* optimiser is:

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

In fact, the optimisations in question were presented in detail in 4. The results collected by the benchmarks are the following.

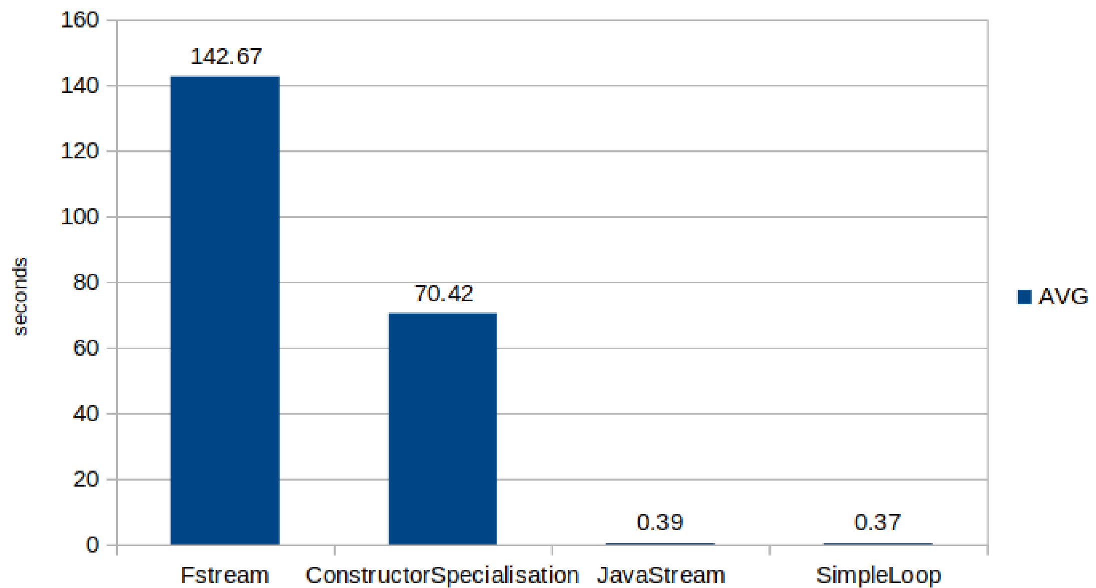


Figure 1: Runtime results for *foldl append*

The results allow us to verify that the optimisation rules applied (and documented in 4) lead to an efficiency gain. The original *FStream* implementation execution time drops from 142 seconds to 70 seconds. Although this version is not the best (as it still does not use *for-each loop*), one can verify that the application of the *GHC* optimisation techniques discussed in Coutts et al. [4] in the form of *source code refactoring* contributes to a major improvement in runtime performance.

The equivalent implementation using the Java Stream API accomplishes a significantly better execution time of 0.39 seconds, which is roughly identical to the 0.37 seconds measured for the final optimised loop version.

7.2 FILTER-MAP

A different example using *filter* and *map* operations was also conceived in order to assess the performance impact of the optimisations with other `FStream` methods.

This example consists in filtering out the `Students` that do not have a grade equal or greater than 95 points. The ones that remain have their grades divided by 10 in order to normalise the value. The initial list has 20.000.000 elements.

The *FStream setting* allows us to write this examples as:

```
Predicate<Student> p = s -> s.grade >= 95;
Function<Student, Student> f =
    s -> new Student(s.name, Math.round((double) s.grade / 10));


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

The obtained results are the following:

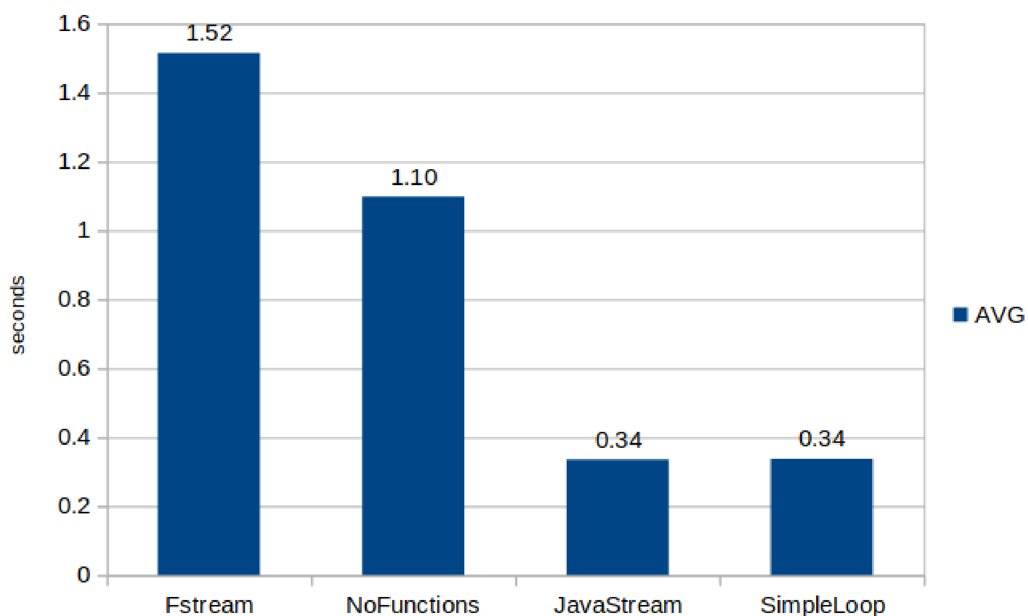


Figure 2: Runtime results for *filter map*

This stream pipeline does not manipulate complex stream states, i.e. the stream does not need to handle objects of types such as `Either` or `Optional` in order to assess which mode it is operating on. Therefore, the *Constructor Specialisation* optimisation is not performed, as there is no need for it.

In the chart, *NoFunctions* represents the final optimised loop (after all Inlines, Case Of Case and Trivial Rewrites) with all `Function` objects removed in order to minimise the overhead.

The original implementation (as seen in the previous code listing) finished its execution after 1.52 seconds.

After all the optimisations were applied, the execution time dropped to 1.10 seconds, which approximately represents a 27% improvement.

If the task in question is implemented with Java Streams or with *for-each* loops, the execution time is around 0.34 seconds.

7.3 ITERATE-ZIP

The calculation of the lines that compose the *Pascal Triangle* can be expressed using the `iterate`, `append`, `zip`, `map` and `take` functions.

The implementation of such solution for the first 3000 lines (`NLINES = 3000`) stands as follows:

```
Function<List<BigInteger>,List<BigInteger>> f1 =
    row ->
        fstream(Arrays.asList(BigInteger.ZERO)).appendfs(fstream(row))
        .zipfs(fstream(row).appendfs(fstream(Arrays.asList(BigInteger.ZERO
            ))))
        .mapfs(p -> p.getX().add(p.getY()))
        .unfstream();

res1 = iterate(f1, l).take(NLINES).unfstream();
```

`zipfs` is an operation that works with complex stream states. However, it restricts the use of the *Constructor Specialisation* rule, as the different modes of operation alternate between each other, making it impossible to split the loop in two parts.

Nevertheless, for this particular example, there is a way to overcome that difficulty and perform an extra optimisation in the end that resembles the effect obtained from the *Constructor Specialisation* rule application.

The two lists that `zipfs` handles are the result of two separate list concatenations. But because these two concatenations yield lists of equal size, we can switch the places of `appendfs` and `zipfs` inside the loop. That way, *append* is the outermost operation and we can then eliminate the creation of all the *Left* and *Right* objects.

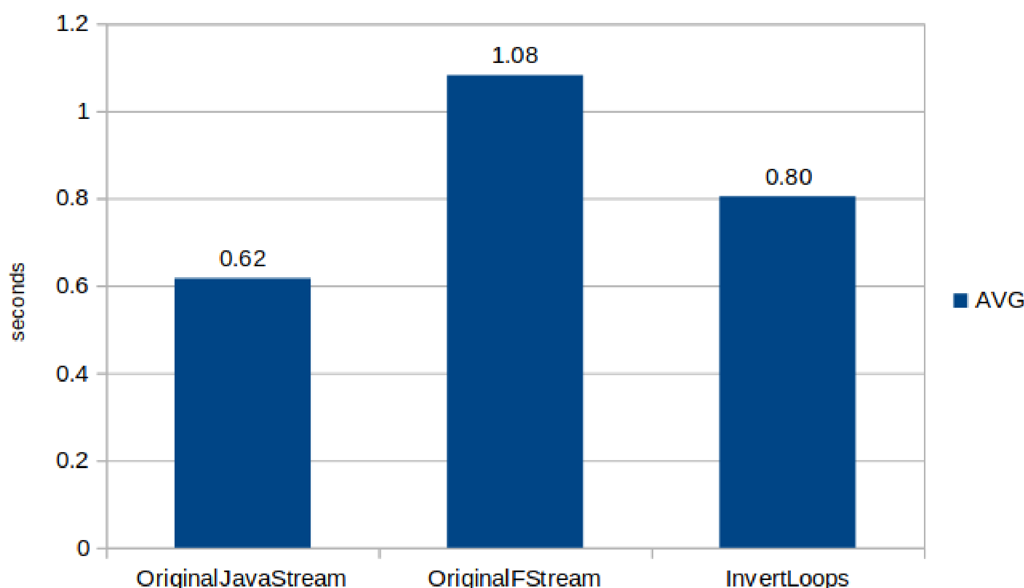


Figure 3: Runtime results for *iterate-zip*

Again, we can see that the transformations have a positive impact in *runtime* performance. The original program takes 1.08 seconds to complete, whereas the final optimised one (called InvertLoops because of the splitting that occurs after the inversion of the operations) takes 0.80, reflecting, approximately, a 26% improvement. Although Java Stream's do not provide a *zip* function implementation, it is still possible to code an equivalent solution by resorting to other methods. The corresponding solution completes after 0.62 seconds.

7.4 FOLDR

Native Java Streams do not currently support a *fold right* like operation.
The `FStream` class implements such a method in different ways:

- The usual recursive `foldr`;
- *foldr as foldl*;
- A loop version derived from a tail recursive definition.

The example chosen to test the performance of the existing alternatives was the calculation of the average of a list of randomly generated numbers ranging from 0 to 499 (inclusive) that gets the numbers that are not even filtered out.

`Foldr`'s stack overflow problems have already been discussed in 3.2. There, it was explained why the *foldr as foldl* alternative took advantage of stepper functions which produced `Skip`s. As such, a *filter* operation was introduced in this example in order to verify that this would happen.

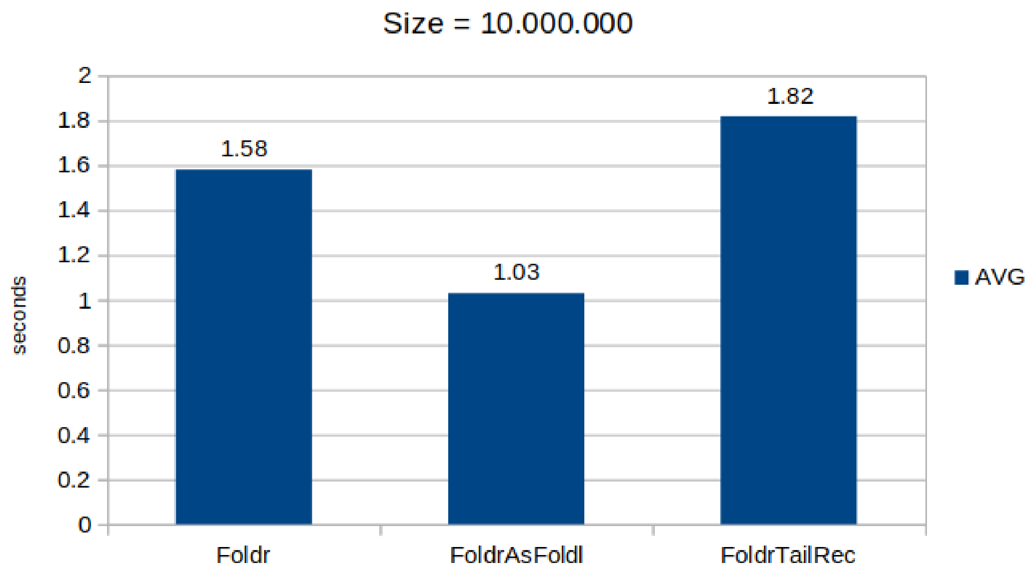


Figure 4: Runtime results for different *foldr* implementations

Running tests for small lists meant that these tests would complete in a very short time. As a consequence, one could not use these values for appropriate comparison, as they placed themselves between 2 and 4 milliseconds.

Thus, in order to be able to produce results for a collection with a large number of elements without running into stack overflow problems, the JVM's internal stack was increased to 1Gb with the `-Xss1G` option. Even `FoldrTailRec` was run under these conditions, even though its implementation does not take advantage of a larger stack size. Nevertheless, it stands to show that the size does not interfere with this version's performance.

The usual recursive version of *foldr* completes the task in 1.58 seconds. The fact that its counterpart `FoldrAsFoldl` finishes in 1.03 seconds goes to show the benefits of "jumping over" `Skip` elements (as discussed in 3.2), which reflect a 35% improvement.

On the other hand, the implementation using *Continuation Passing Style* did worse, averaging 1.82 seconds. Although this version has the ability to produce results for lists containing any amount of elements (provided they all fit in memory), it does so with a penalty. The 77% decrease in performance from 1.03 to 1.82 shows exactly that: the cost of having to handle all those *Continuation* objects.

7.5 FIBONACCI

The adaptation of the *FStream setting* to binary trees was presented in 3.3.

A solution for calculating *Fibonacci* numbers can be expressed based on binary trees (as seen on 6.1).

In this section, we explore two different perspectives. One of them is what has been seen so far: applying the transformations and check if there are any improvements. The other one consists of comparing different program versions for calculating these *Fibonacci* numbers and analysing the differences in execution time between them.

7.5.1 *Applying transformations*

An implementation for the calculation of the *Fibonacci* function can be expressed as an *unfold* followed by a *fold*.

```
Function<Integer, Either<BigInteger, Pair<Integer,Integer>>> g = x -> {
    if (x == 0){
        return new Left<>(BigInteger.ZERO);
    }
    else if(x == 1){
        return new Left<>(BigInteger.ONE);
    }
    else{
        return new Right<>(new Pair<>(x-1, x-2));
    }
};

BiFunction<BigInteger, BigInteger, BigInteger> sum = (x, y) -> x.add(y);

BigInteger res = unfoldrBT(g, 38).foldBTTailRec(sum, Function.identity());
```

In this case, the *fold* that was used was the version implemented with *Continuation Passing Style* because it is the one that has a code structure more susceptible to have the optimisations applied to it.

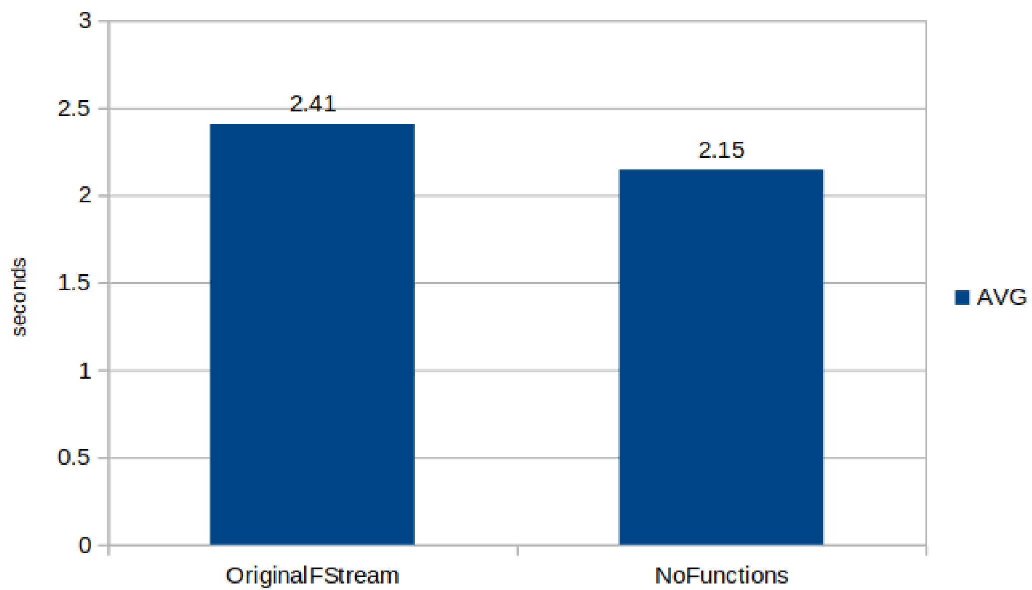


Figure 5: Runtime results for *Fibonacci* before and after optimisations

As illustrated by the chart, the original `FStream` version with no optimisations finishes in 2.41 seconds.

After the transformations and the removal of all `Function` objects, the execution time drops to 2.15 seconds, depicting a 10% gain.

In contrast to the examples so far, results for an implementation using Java Streams are not presented. This is because Java Streams are intended to work over the *Collections* the language has to offer. In this particular case, `FStreams` are working over binary tree structures, thus the absence of a native Java Stream alternative.

7.5.2 *Version comparison*

Fibonacci numbers can be calculted through many different ways. With this in mind, several solutions were implemented in order to better understand the differences between each other in terms of *runtime*. The implementations in question are:

- Recursive: an intuitive and straightforward implementation;
- Recursive fold: similar to the one presented in the previous section, but with a recursive implementation for the final *fold*;
- Anamorphism and Catamorphism (no fusion): explicitly producing the binary tree and then consuming it;
- Hylomorphism (fusion): similar to the previous one, but elements get consumed as they are produced.

- Iterative: using a simple *for* loop;
- Tuple: uses an auxiliary tuple to return the solution for *n-1* and *n* in order to avoid repeating the calculations;

The first four implementations were tested for *n = 38* and the collected results were the following.

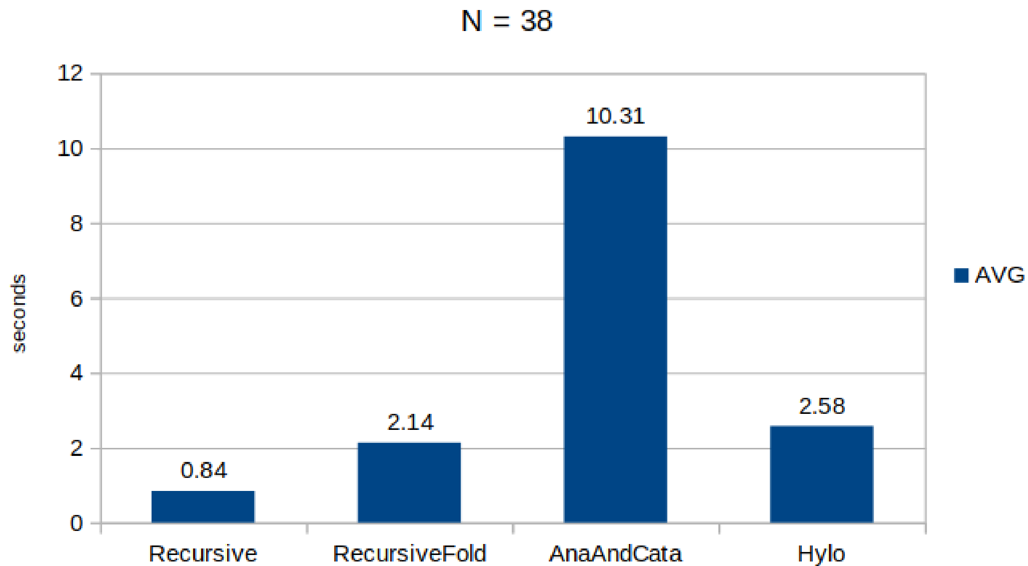


Figure 6: Runtime results for *recursive*, *recursive fold*, *anamorphism/catamorphism* and *hylomorphism*

The recursive version, although having the disadvantage of running into a stack overflow for bigger *n* values, is the one that comes on top among these four alternatives, taking approximately 0.84 seconds to compute *fib(38)*.

The *fold.unfold* version based on a recursive implementation takes 2.14 seconds to compute the same result, which makes for a 154% increase in *runtime*.

Explicitly building the produced structure and then consuming it is, as expected, the worst implementation in terms of execution time. At 10.31 seconds, it takes approximately 12x more time to complete than the most efficient one.

Finally, an implementation based on a hylomorphism, where elements are consumed as they are produced, reduces the previous result to 2.58 seconds, which means a 75% improvement. It is important to note that this solution is very similar to the one of *RecursiveFold*, except that it is able to compute bigger *n* values (as it does not have to deal with the increasing stack size originated by the recursive function calls) and its code is prone to optimisations (as already mentioned).

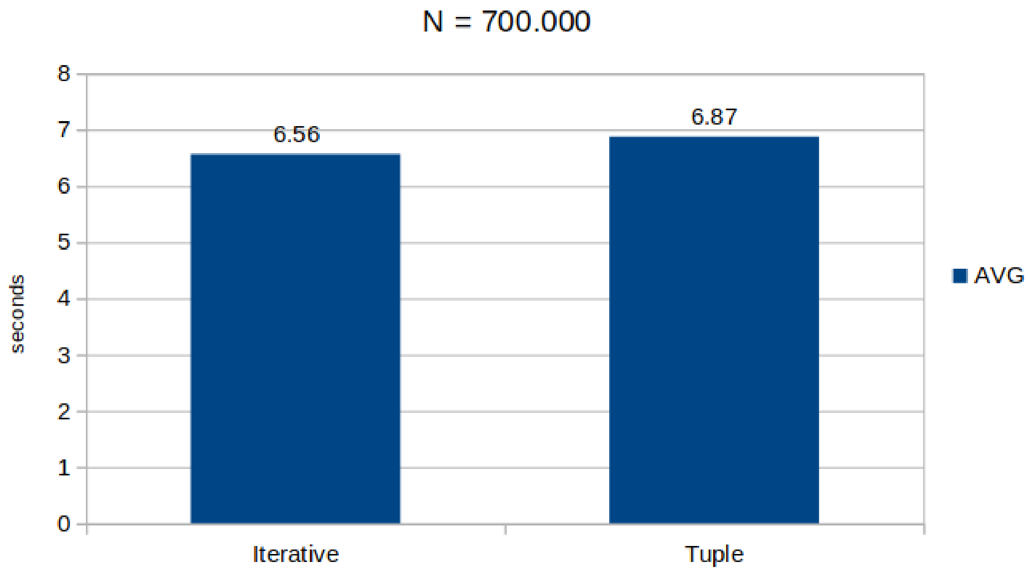


Figure 7: Runtime results for *iterative* and *tuple*

The two last implementations - iterative and tuple - are the most efficient ones. In fact, a significantly greater *n* value of 700.000 had to be used in order to obtain comparable execution time results.

As the implementation that makes use of *tuples* is a recursive one, the JVM internal stack size needed to be increased to 1Gb through the `-Xss1G` option.

The two solutions present very similar performance, separated by a 5% difference with the *for* loop implementation finishing after 6.56 seconds and the *tuple* based one after 6.87 seconds.

Still, as stated previously for another set of implementations, there are stack overflow errors that are bound to happen as the value of *n* increases, which the *iterative* implementation is able to overcome but the *tupled* solution suffers from.

7.6 Initial experiments

Some preliminary tests were made in order to better understand the behaviour of the native implementation of Java streams before and after fusion.

For now, the transformation was applied manually.

7.6.1 *Experiment setup*

The focus for the experiments being documented was execution time.

As the tests being performed targeted very specific scenarios of stream usage, there are some risks associated with the isolated execution of the examples in question that needed to be addressed.

More specifically, the JVM might apply some optimizations to the code segment being executed. However, in a real world scenario, these optimisations might not be performed if that part of the program is integrated in a larger application.

As such, while performing benchmarks of this kind, we need to concern about several issues, such as:

- Loop Optimisations
- Dead Code Elimination
- Constant Folding
- JVM Warmup

In order to compensate for these difficulties, in addition to what we want to measure, the benchmark needs to implement preventive measures to mitigate unwanted effects.

Fortunately, there are tools that assist with this kind of requirements.

One example is *Java Microbenchmark Harness (JMH)*.

With the placement of some convenient annotations, the framework will benchmark the code we intend in an appropriate way. This way, we can keep our focus in the actual code to be benchmarked and thus saving time in the setup.

The collection used for the tests consisted of an *ArrayList* of *String*, with each of the elements having the form *n_word* where *n* is a random integer.

In order to test the effects of fusion, two *filter* alternatives for the same task were conceived. One had the predicates to be tested *chained* together and the other one had them *merged*.

The operation is question consisted of calculating the number of elements in the list that complied to two predicates:

- *n* was greater than 100,000,000
- the *String*'s length was less than 15

Therefore, the chained alternative was equivalent to:

```
list.stream().filter(s -> Integer.parseInt(s.split("_")[0]) > 100000000)
             .filter(s.length() < 15)
             .count();
```

And the merged alternative was equivalent to:

```
list.stream().filter(s -> Integer.parseInt(s.split("_")[0]) > 100000000
                     && s.length() < 15)
              .count();
```

Different collection sizes were considered: 5,000,000, 10,000,000, 20,000,000, 40,000,000.
Each case had 5 *warmup* iterations of 20 seconds each.
For the actual measurement, 8 iterations of 20 seconds each were executed.

7.6.2 *Results*

For the experiment described in the previous section, the following results were obtained:

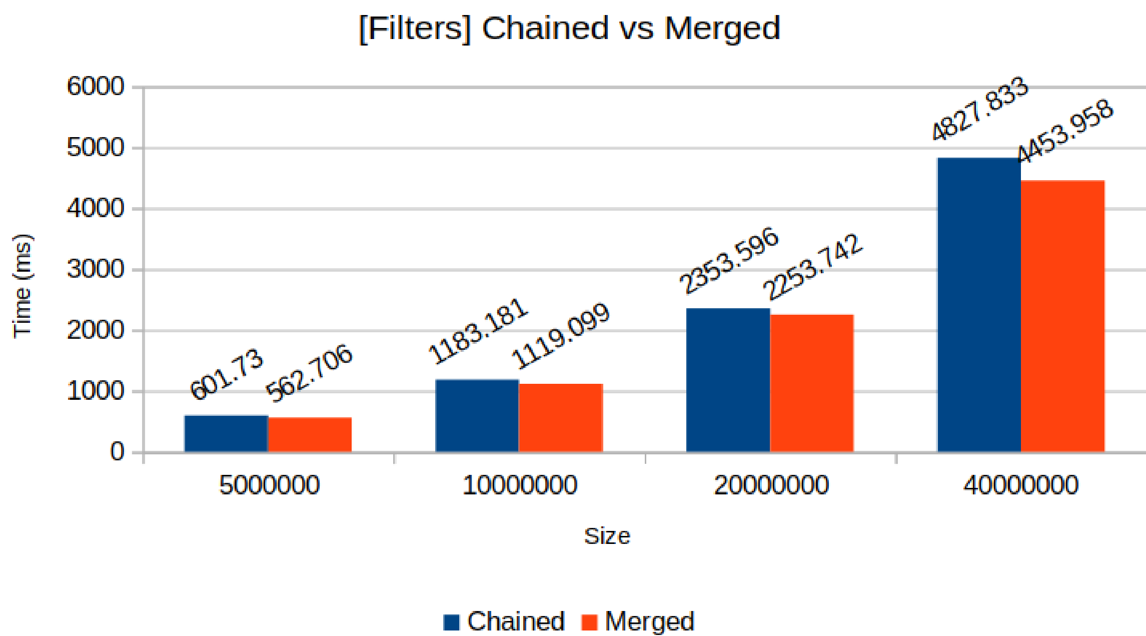


Figure 8: Filter: chained vs merged results

As one can see, the merged alternative is faster than the option with chained predicates. If we calculate the percent improvement of the merged version over the chained version:

- 5,000,000 elements: 6.5%
- 10,000,000 elements: 5.4%
- 20,000,000 elements: 4.2%

- 40,000,000 elements: 7.7%

We can end up considering that there is a general 6% improvement when fusion is applied.

7.6.3 *Discussion*

The improvement verified after applying fusion was expected.

What's at the heart of this improvement is not the number of times each predicate gets tested, but the number of traversals done on the data structures.

Because the operations are chained, there is an intermediate structure being produced after the execution of the first filter. For that intermediate structure, the elements that did not comply to the first predicate have already been removed. Nevertheless, this structure (a newly produced stream) still needs to get traversed in order to check if its elements obey the second predicate.

However, with fusion (mentioned as "merged"), the two predicates are tested for each element right after the other. Therefore, we avoid the creation of an intermediate structure and, consequently, an extra unnecessary traversal.

Although the outcomes show an improvement, these gains are not significant. Therefore, at this time, the results are not conclusive and more tests are going to be performed.

8

CONCLUSION/FUTURE WORK

8.1 CONCLUSIONS

Throughout the years, programming languages have come up with new mechanisms that allow programmers to abstract more complex ideas into simple instructions. *Java Streams* are an example of such a mechanism, allowing for an easier manipulation of *Java collections*. Together with *lambda* expressions, programmers can express recursion patterns in a very intuitive way. However, these abstractions may lead to performance issues. Chaining several *higher order functions* can cause a program to perform extra unnecessary traversals and operations if optimisation techniques like *fusion* are not implemented.

In chapter 2, several optimisation techniques were covered. Techniques like *deforestation* and *short-cut fusion* aimed to eliminate intermediate structures that were inevitably created as a way to "glue" different functions together. Other approaches, like *circular program calculation*, focused on converting algorithms which performed multiple traversals to programs performing a single one. Ultimately, *stream fusion*, in which this thesis is heavily based on, accomplishes both. By rewriting Haskell's List library functions in order to adapt them to the *Stream* setting and, together with that, providing a set of compiler optimisation rules, this approach accomplishes some kind of automation when it comes to fusion. Automating this final step is something that previous techniques have missing. In a similar way, the *Hylo System* also tries to perform these transformations automatically by extracting the recursive structure of programs and performing fusion through the application of transformation laws.

In order to explore the approach taken by *Stream Fusion* in a Java context, that setting has to be adapted and implemented in Java. The representation of the *Stream* and *Step* datatypes had to be mapped to a more Object Oriented setting and each function of the *Stream* library had to have its implementation converted to the most possibly equivalent Java method. Haskell's functional essence makes recursion a natural "component" of the

language. As such, many patterns are implemented recursively, like *folds*. Although the Java language allows for the use of recursion, it is not the most efficient mechanism. Recursive calls rapidly fill up the stack and, in order to overcome some of these difficulties, alternative ways to express *foldr* had to be explored. As such, the reader may have noticed that in 3.2.3, *Continuation Passing Style* was given a lot of attention if we consider that the main research task described in this document did not initially focus on something like that. This was something that emerged from a difficulty which sprouted during the development process of the *FStream setting*. Moreover, with the adaptation of the framework to other data structures, this programming style also helped overcoming problems of similar nature.

After having the *FStream setting* all set up, it was now possible to write programs by chaining several higher order functions. However, a considerable amount of object allocations had been introduced in order to eliminate intermediate data structures. Therefore, the code that got executed at that point was not optimal. [4] actually describe this scenario and explain that a lot of the optimisation process is carried out by GHC. In order to reproduce the different optimisation rules already implemented in GHC, the examples implemented to demonstrate the use of *FStreams* were refactored. That way, the effect of the different optimisation steps was recreated and the ultimate goal of removing unrequired allocations was accomplished.

Although the optimisations are not automatic, if we analyse the structure of the different programs, a number of similarities are found. Thus, general templates can be extracted, which assists the optimisation process. Depending on the type of functions which compose the stream pipeline, different templates need to be considered, which ultimately affect the optimisation rules that are applied.

Although Java already has the notion of *stream* natively present in its implementation, the *FStream setting* brings something new to the table: expressiveness. By implementing functions not supported (at least currently) by native Java streams and making it feasible to work over other data structures like binary leaf trees, it makes it possible to implement certain algorithms in a more expressive way. Calculating *fibonacci* numbers through the use of binary leaf trees is one example, as shown in 6.1.

Like every *software* piece, the framework developed throughout this thesis was subject to performance tests. The main goal of these tests was to assess the impact that the source code transformations had on the execution time of several example programs. Those numbers were put side by side with *run time* results for equivalent versions of those same programs,

but using Java Streams and normal *for* loops.

Chapter 8 describes one of the first tasks that was carried out during the beginning of this thesis. The goal was to verify the degree of improvement that was possible to achieve by simply merging each parameter of two higher order functions of the same type into only one.

8.2 PROSPECT FOR FUTURE WORK

Ideally, fusion should be accomplished through an automated optimisation process, similar to what Coutts et al. [4] and Onoue et al. [10] try to achieve. Therefore, future work could focus on developing a tool that automatically performs the described refactorings in 4.

FStreams performance (without optimisations) falls behind Java Streams. Athough this gap tries to be shortened through the transformations previously mentioned, it would be interesting to explore what improvements could be achieved by actually changing the implementation of the setting.

In addition to making it possible to abstract complex recursion patterns into pipelines of simple and clear higher order functions, the concept of stream also simplifies the adaptation of sequential implementations of algorithms to parallel ones. As such, making the necessary adaptations to *FStream* in order to have it support parallelism constitutes one of the main features for future work.

BIBLIOGRAPHY

[1] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, Oct 1984. ISSN 1432-0525. doi: 10.1007/BF00264249. URL https://doi.org/10.1007/BF00264249.

[2] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. Towards a green ranking for programming languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, SBLP 2017, pages 7:1–7:8, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5389-2. doi: 10.1145/3125374.3125382. URL http://doi.acm.org/10.1145/3125374.3125382.

[3] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. Energyware analysis. In *Proceedings of the Seventh Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, August 27-30, 2018.*, 2018. URL https://perun.pmf.uns.ac.rs/pracner/down/sqamia2018/paper-per.pdf.

[4] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP'07*, 2007.

[5] João Paulo Fernandes, Alberto Pardo, and João Saraiva. A shortcut fusion rule for circular program calculation. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 95–106, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5. doi: 10.1145/1291201.1291216. URL http://doi.acm.org/10.1145/1291201.1291216.

[6] João Paulo Fernandes, João Saraiva, Daniel Seidel, and Janis Voigtländer. Strictification of circular programs. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 131–140, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0485-6. doi: 10.1145/1929501.1929526. URL http://doi.acm.org/10.1145/1929501.1929526.

[7] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: 10.1145/165180.165214. URL http://doi.acm.org/10.1145/165180.165214.

[8] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[9] Pedro Martins, João Paulo Fernandes, and João Saraiva. *Zipper-Based Modular and Deforested Computations*, pages 407–427. Springer International Publishing, Cham, 2015. ISBN 978-3-319-15940-9. doi: 10.1007/978-3-319-15940-9_10. URL https://doi.org/10.1007/978-3-319-15940-9_10.

[10] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. *A Calculational Fusion System HYLO*, pages 76–106. Springer US, Boston, MA, 1997. ISBN 978-0-387-35264-0. doi: 10.1007/978-0-387-35264-0_4. URL https://doi.org/10.1007/978-0-387-35264-0_4.

[11] Alberto Pardo, João Paulo Fernandes, and João Saraiva. Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In Germán Puebla and Germán Vidal, editors, *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, pages 81–90. ACM, 2009. ISBN 978-1-60558-327-3. doi: 10.1145/1480945.1480958. URL http://doi.acm.org/10.1145/1480945.1480958.

[12] Alberto Pardo, João Paulo Fernandes, and João Saraiva. Shortcut fusion rules for the derivation of circular and higher-order programs. *Higher-Order and Symbolic Computation*, 24(1):115–149, Jun 2011. ISSN 1573-0557. doi: 10.1007/s10990-011-9076-x. URL https://doi.org/10.1007/s10990-011-9076-x.

[13] Alberto Pardo, João Paulo Fernandes, and João Saraiva. Multiple intermediate structure deforestation by shortcut fusion. In André Rauber Du Bois and Phil Trinder, editors, *Programming Languages*, pages 120–134, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40922-6.

[14] Alberto Pardo, João Paulo Fernandes, and João Saraiva. Multiple intermediate structure deforestation by shortcut fusion. *Science of Computer Programming*, 132:77 – 95, 2016. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2016.07.004. URL http://www.sciencedirect.com/science/article/pii/S0167642316300880. Selected and extended papers from SBLP 2013.

[15] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5525-4. doi: 10.1145/3136014.3136031. URL http://doi.acm.org/10.1145/3136014.3136031.

[16] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6 (3):233–247, Nov 1993. ISSN 1573-0557. doi: 10.1007/BF01019459. URL https://doi.org/10.1007/BF01019459.

[17] João Saraiva and Doaitse Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction, CC/ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999.

[18] Gerald J. Sussman and Guy L. Steele, Jr. An interpreter for extended lambda calculus. Technical report, Cambridge, MA, USA, 1975.

[19] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231 – 248, 1990. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(90)90147-A. URL http://www.sciencedirect.com/science/article/pii/030439759090147A.

This work is funded by ERDF - European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the FCT - Foundation for Science and Technology within the project *FCOMP-01-0124-FEDER-020484* and grant ref. *BI2-2017_PTDC/EEI-ESS/5341/2014_UMINHO*.