# Semester Project Report

## Advancing Algebraic Reasoning for Scala


*Author:*
Lucien Michaël Iseli

*Professor:*
Viktor Kunčak
*Supervisor:*
Romain Ruetschi


June 15, 2019

# Advancing Algebraic Reasoning for Scala

Lucien Michaël Iseli

June 15th, 2019

## Contents

# Introduction

Stainless is a tool that verifies laws and properties statically for Scala code, it supports a good chunk of functional Scala, however not everything is supported yet. It has its own implementations of List, Set, Map, etc... The goal of this project is to write code for Stainless, to have a bigger example project written with Stainless while also extending its library and enhancing it. The goal is to define algebraic structures and have a bigger project example inside Stainless, and improving the library.

# Laws

## Comparison

The first laws we wanted to define were the ones used to compare objects together. In order to define comparison we have a hierarchical structure containing `Equality`, `Partial order` and `Total order`. All these relations have mathematical properties that must hold and Stainless is a good tool to verify such things.

### Equality

Mathematical equality is defined as an equivalence relation.[1] Thus any equality relation must verify 3 properties:

- Reflexive:
  $\forall x : x = x$
- Symmetric:
  $\forall x, y : x = y \iff y = x$
- Transitive:
  $\forall x, y, z : x = y \land y = z \implies x = z$

Then in stainless we can translate these 3 properties into 3 laws that will be verified. Since classes in Stainless inherit laws, any user-defined equality relation that extends this class will be verified by Stainless statically.

[1] https://en.wikipedia.org/wiki/Equality_(mathematics)#Basic_properties

```
abstract class Equality[A] {
  def eqv(x: A, y: A): Boolean

  @law
  def law_reflexive_equality(x: A) = {
    eqv(x, x)
  }

  @law
  def law_symmetric_equality(x: A, y: A) = {
    eqv(x, y) == eqv(y, x)
  }

  @law
  def law_transitive_equality(x: A, y: A, z: A) = {
    (eqv(x, y) && eqv(y, z)) ==> eqv(x, z)
  }
}
```

Given the above laws, to define a new Equality relation for any object it will be sufficient to extend the Equality class by writing the eqv method and Stainless will verify whether these relations hold true.

### Partial order

Partial order, as the name implies, defines an ordering between elements however it is partial. Thus some elements cannot be compared together, they are not the same elements but none of them must go before the other. The order is defined by the $\leq$ relation. Partial order extends `Equality` in this hierarchy and must verify 3 properties:[2]

- Reflexive:
  $\forall x : x \leq x$
- Antisymmetry:
  $\forall x, y : x \leq y \land y \leq x \implies x = y$
- Transitivity:
  $\forall x, y, z : x \leq y \land y \leq z \implies x \leq z$

Which, in Stainless, translates to:

[2] https://en.wikipedia.org/wiki/Partially_ordered_set#Formal_definition

```
abstract class PartialOrder[A] extends Equality[A] {
  def lteqv(x: A, y: A): Boolean

  @law
  def law_reflexive_partial_order(x: A) = {
    lteqv(x, x)
  }

  @law
  def law_antisymmetric_partial_order(x: A, y: A) = {
    (lteqv(x, y) && lteqv(y, x)) ==> eqv(x, y)
  }

  @law
  def law_transitive_partial_order(x: A, y: A, z: A) = {
    (lteqv(x, y) && lteqv(y, z)) ==> lteqv(x, z)
  }
}
```

### Total order

Following the same design we can define `Total order`, which is basically `Partial order` but all elements have to be comparable. `Total order` is also defined by the $\leq$ relation and since we extend `Partial order` we only have to add this property, called the connex property:[3]

$\forall x, y : x \leq y \vee y \leq x$

Note that both $x \leq y$ and $y \leq x$ can be true which means, using the antisymmetric property, that $x = y$.

Due to inheritance the class definition is small:

```
abstract class TotalOrder[A] extends PartialOrder[A] {
  @law
  def law_connex_total_order(x: A, y: A) = {
    lteqv(x, y) || lteqv(y, x)
  }
}
```

### Example: Total order

Here are two examples of verified implementations of `Total order` (which due to inheritance also define both a verified `Partial order` and a verified `Equality`. First `BigInt`:

[3] https://en.wikipedia.org/wiki/Total_order

```
case class BigIntTotalOrder() extends TotalOrder[BigInt] {
  def eqv(x: BigInt, y: BigInt): Boolean = {
    x == y
  }

  def lteqv(x: BigInt, y: BigInt): Boolean = {
    x <= y
  }
}
```

In this example, Stainless manages to prove everything on its own because Stainless has an inner implementation of `BigInt` and thus it knows many properties about `BigInts`.

Second, `Nat` (Natural numbers, defined here):

*The code is a bit too long to insert here but it can be found here*[4].

This example is slightly more involved, as Stainless doesn't manage to prove it on its own but it's still relatively easy. The only thing I helped with is making Stainless use induction. For the 3 properties the only thing the proofs helpers do is saying that the property holds for x = Succ(n) because it holds for n and then Stainless does the rest. Here is one of the methods, as an example to show the process:

```
def law_connex_Nat(x: Nat, y: Nat): Boolean = {
  (x <= y || y <= x) because {
    (x, y) match {
      case (Succ(n), Succ(m)) =>
        assert(law_connex_Nat(n, m))
        check(x <= y || y <= x)
      case _ => true
    }
  }
}.holds
```

I only tell Stainless that the property is true because it's true for `n` and `m` and it is enough. The other methods are very similar.

### Example: Sorting

Now that we have total orders we can define something more meaningful that uses them. I implemented an insertion sort and a merge sort, both verified (*the code is available here*):

- Insertion sort[5]
- Merge sort[6]

[4] https://github.com/Gorzen/stainless/blob/sorting-project/lucien/TotalOrderNat.scala
[5] https://github.com/Gorzen/stainless/blob/sorting-project/lucien/InsertionSort.scala
[6] https://github.com/Gorzen/stainless/blob/sorting-project/lucien/MergeSort.scala

The sort method has the following definition:

```
def sort[T](list: List[T])(implicit comparator: TotalOrder[T]): List[T] = {
  // Implementation...
} ensuring { res =>
  bag(list) == bag(res) &&
  isSorted(res)
}
```

Since `Bag` is a multiset, `bag(res)` = `bag(list)` $\implies$ `res` and `list` have exactly the same content; same elements repeated the same number of times. Plus, we have a postcondition that the list is sorted, thus ensuring that the returned list has the same content and is sorted.

I had to help Stainless with the proofs a little bit but nothing too fancy, see the code for further details. I had to put postconditions on different methods and a few assertion to help Stainless but that's it. One issue I had was that in my first implementation merge sort had basically the same runtime as insertion sort. The problem was that the functional List abstraction Scala has makes splitting a List not efficient at all, $O(n)$. Indeed, you have to traverse the list to split it somewhere since it's basically a linked list. However, after improving the splitting and making sure that the proofs still hold, merge sort was eventually faster. *You can see some runtimes here.*

## Semigroups and monoids

Monoids define an interesting algebraic structure and will be essential in order to define the fold method used in this case study.

### Semigroup

Monoids are a subset of semigroups, thus following the same hierarchical structure as the one used for comparison, I implemented `SemiGroup` and `Monoid` independently.

Semigroups and monoids define a type and an operation that combines two elements of the same type returning an element of the same type. It is a binary operation. Semigroups must verify one property: assosciativity. *The formal definition can be found here.*[7]

A semigroup is a set $S$ and a function $\oplus : S \times S \to S$ that satisfies the assosciative property:

$\forall a, b, c \in S : (a \oplus b) \oplus c = a \oplus (b \oplus c)$

As usual mathematical properties translate really smoothly to Scala/Stainless:

[7] https://en.wikipedia.org/wiki/Semigroup#Definition

```
abstract class SemiGroup[A]{
  def append(x: A, y: A): A

  @law
  def law_associativity(x: A, y: A, z: A) = {
    append(x, append(y, z)) == append(append(x, y), z)
  }
}
```

Where `append` defines the binary operation $\oplus$.

### Monoid

Monoids are semigroups, thus they are defined by a set and a binary operation. However for a semigroup to be a monoid it must also satisfy some other properties. Monoids have an empty element as well as left and right identity:[8]

$\exists e \in S \mid \forall a \in S : e \oplus a = a \oplus e = a$

```
abstract class Monoid[A] extends SemiGroup[A]{
  def empty: A

  @law
  def law_leftIdentity(x: A) = {
    append(empty, x) == x
  }

  @law
  def law_rightIdentity(x: A) = {
    append(x, empty) == x
  }
}
```

### Examples

I wrote some examples for `BigInt`, `List` and `Option`. As Stainless didn't need too much help the code doesn't really need any further explanation. The code can be found here:

- BigInt, addition[9]
- BigInt, multiplication[10]
- List[A], concatenation[11]
- Option[A], given a Monoid[A][12]

[8] https://en.wikipedia.org/wiki/Monoid#Definition
[9] https://github.com/Gorzen/stainless/blob/sorting-project/lucien/MonoidBigInt.scala
[10] https://github.com/Gorzen/stainless/blob/sorting-project/lucien/MonoidBigInt.scala
[11] https://github.com/Gorzen/stainless/blob/sorting-project/lucien/MonoidList.scala
[12] https://github.com/Gorzen/stainless/blob/sorting-project/lucien/MonoidOption.scala

# Folding

`Fold` is a very important higher-order function in functional programming, it combines every element of a list and outputs a result. It is useful because it is very versatile and can be parallelized. Plus, it has interesting properties when it is defined on monoids as we'll see later on.

## Lists

As said previously, monoids are really handy for folding operations and we will need a fold operation. The fold on List is defined as such, using the `Monoid` we defined earlier:

```
def fold[A](xs: List[A])(implicit M: Monoid[A]): A = {
  xs.foldLeft(M.empty)(M.append)
}
```

Here are the definitions of foldLeft and foldRight (*defined on* `List[T]`):

```
def foldLeft[R](z: R)(f: (R,T) => R): R = this match {
  case Nil() => z
  case Cons(h,t) => t.foldLeft(f(z,h))(f)
}

def foldRight[R](z: R)(f: (T,R) => R): R = this match {
  case Nil() => z
  case Cons(h, t) => f(h, t.foldRight(z)(f))
}
```

I added some basic checks to verify that this fold does what we expect (sum elements correctly, etc). Plus, I proved a property that will be really handy and is really interesting because it comes from the fact that we use monoids. It is that foldLeft == foldRight. Why is it handy to go from foldLeft to foldRight? Because Stainless is way better at reasoning about foldRight rather than foldLeft. FoldRight simply calls itself on the list with one element less and then calls the append function on it, foldLeft's recursive call has two different parameters including one on which induction doesn't necessarily make sense. This makes induction proofs way easier to write using foldRights.

The proof is interesting and required some thinking to write. Here it is (with base cases omitted, you can look at them in the appendix here):

Case xs == y1 :: y2 :: ys $\implies$ xs has $\geq 2$ elements.

$$xs.foldLeft(M.empty)(M.append) \quad (1)$$
$$\textbf{xs definition} \implies \quad (2)$$
$$(y1 :: y2 :: ys).foldLeft(M.empty)(M.append) \quad (3)$$
$$\textbf{foldLeft definition} \implies \quad (4)$$
$$(y2 :: ys).foldLeft(M.append(M.empty, y1))(M.append) \quad (5)$$
$$\textbf{left identity} \implies \quad (6)$$
$$(y2 :: ys).foldLeft(y1)(M.append) \quad (7)$$
$$\textbf{foldLeft definition} \implies \quad (8)$$
$$ys.foldLeft(M.append(y1, y2))(M.append) \quad (9)$$
$$\textbf{left identity} \implies \quad (10)$$
$$ys.foldLeft(M.append(M.empty, M.append(y1, y2)))(M.append) \quad (11)$$
$$\textbf{foldLeft definition} \implies \quad (12)$$
$$(M.append(y1, y2) :: ys).foldLeft(M.empty)(M.append) \quad (13)$$
$$\textbf{induction hypothesis, since the size of the list is smaller} \implies \quad (14)$$
$$(M.append(y1, y2) :: ys).foldRight(M.empty)(M.append) \quad (15)$$
$$\textbf{foldRight definition} \implies \quad (16)$$
$$M.append(M.append(y1, y2), ys.foldRight(M.empty)(M.append) \quad (17)$$
$$\textbf{assosciativity} \implies \quad (18)$$
$$M.append(y1, M.append(y2, ys.foldRight(M.empty)(M.append) \quad (19)$$
$$\textbf{foldRight definition} \implies \quad (20)$$
$$M.append(y1, (y2 :: ys).foldRight(M.empty)(M.append)) \quad (21)$$
$$\textbf{foldRight definition} \implies \quad (22)$$
$$(y1 :: y2 :: ys).foldRight(M.empty)(M.append) \quad (23)$$
$$\textbf{xs definition} \implies \quad (24)$$
$$xs.foldRight(M.empty)(M.append) \quad (25)$$

Very nice, now we can easily translate it to Scala code and Stainless validates it! Notice that for the proof to work we need assosciativity + identity, the only properties monoids have; nothing more, nothing less.

## ConcRopes

Now we need to define a folding method on ConcRope when given a monoid in order to compare it to List in our case study.

*Reminder:*

ConcRope is a tree-like structure composed of 4 different nodes[13]:

[13] http://aleksandar-prokopec.com/resources/docs/lcpc-conc-trees.pdf

```
case class Empty[T]() extends Conc[T]
case class Single[T](x: T) extends Conc[T]
case class CC[T](left: Conc[T], right: Conc[T]) extends Conc[T]
case class Append[T](left: Conc[T], right: Conc[T]) extends Conc[T]
```

The tree is balanced, the level difference between the two children cannot be greater than 1. Append is a special node that provides an amortized $O(1)$ append and prepend methods. CC is the usual tree object.

Thus we can define the fold method in the following way:

```
def foldSequential[A](xs: Conc[A])(implicit M: Monoid[A]): A = {
  xs match {
    case Empty() => M.empty
    case Single(x) => x // Thanks to left identity we can simply return x
    case CC(left, right) =>
        M.append(foldSequential(left), foldSequential(right))
    case Append(left, right) =>
        M.append(foldSequential(left), foldSequential(right))
  }
}
```

This method is sequential as the name implies, and ConcRopes are designed to be easily and efficiently parallelized. Indeed this method is easily parallelized. We compute the fold of both children and then append them together but we can compute both folds in parallel since they are independent and then append the results.

Now, we need to verify that this fold method indeed does what we expect. To do this I proved:

```
def proof[A](xs: Conc[A]): Boolean = {
    fold(xs.toList) == fold(xs)
}.holds
```

Namely, folding the ConcRope or the List defined by the ConcRope yields the same result.

This proof is a little long you can take a look at it here[14]. But I think it deserves some explanation, so here is an overview of how it works. The proof is yet again an inductive one, we prove base cases (`Empty` and `Single`) and then for `CC` and `Append` we roughly do the following:

[14] https://github.com/Gorzen/stainless/blob/sorting-project/lucien/FoldMapConcRope.scala

```
xs = CC(left, right) || Append(left, right)

// Want to prove:
fold(xs) == fold(xs.toList)

//Induction hypothesis
fold(left) == fold(left.toList)
fold(right) == fold(right.toList)

xs.toList == left.toList ++ right.toList

fold(xs) == M.append(fold(left), fold(right))
    == M.append(fold(left.toList), fold(right.toList))

// ==> need to prove:
M.append(fold(left.toList), fold(right.toList)) ==
    fold(left.toList ++ right.toList)
```

Now we need to prove this property. Its proof is interesting and uses the proof I showed earlier (foldLeft == foldRight). Thus, I thought it could be insightful to add it here. I wrote it in latex but once again the mapping from mathematical reasoning to Stainless is pretty straightforward.

We want to show:

```
M.append(fold(xs), fold(ys)) == fold(xs ++ ys)
```

*Note: This means that the function fold: List[A] → A is an homomorphism for the monoid ++ on List[A] and any monoid on A*[15] *!*

The base case is omitted here, but is in the code, this means that:

xs == z :: zs $\implies$ xs has $\geq 1$ element.

[15] https://en.wikipedia.org/wiki/Monoid#Monoid_homomorphisms

$$M.append(fold(xs), fold(ys)) \quad (1)$$
$$\textbf{xs definition} \implies \quad (2)$$
$$M.append(fold(z :: zs), fold(ys)) \quad (3)$$
$$\textbf{fold definition} \implies \quad (4)$$
$$M.append((z :: zs).foldLeft(M.empty)(M.append), fold(ys)) \quad (5)$$
$$\textbf{foldLeft == foldRight on monoid} \implies \quad (6)$$
$$M.append((z :: zs).foldRight(M.empty)(M.append), fold(ys)) \quad (7)$$
$$\textbf{foldRight definition} \implies \quad (8)$$
$$M.append(M.append(z, zs.foldRight(M.empty)(M.append)), fold(ys)) \quad (9)$$
$$\textbf{assosciativity} \implies \quad (10)$$
$$M.append(z, M.append(zs.foldRight(M.empty)(M.append), fold(ys))) \quad (11)$$
$$\textbf{foldLeft == foldRight on monoid} \implies \quad (12)$$
$$M.append(z, M.append(zs.foldLeft(M.empty)(M.append), fold(ys))) \quad (13)$$
$$\textbf{fold definition} \implies \quad (14)$$
$$M.append(z, M.append(fold(zs), fold(ys))) \quad (15)$$
$$\textbf{induction hypothesis, since size of zs} < \textbf{size of xs} \implies \quad (16)$$
$$M.append(z, fold(zs + +ys)) \quad (17)$$
$$\textbf{fold definition} \implies \quad (18)$$
$$M.append(z, (zs + +ys).foldLeft(M.empty)(M.append)) \quad (19)$$
$$\textbf{foldLeft == foldRight on monoid} \implies \quad (20)$$
$$M.append(z, (zs + +ys).foldRight(M.empty)(M.append)) \quad (21)$$
$$\textbf{foldRight definition} \implies \quad (22)$$
$$(z :: zs + +ys).foldRight(M.empty)(M.append) \quad (23)$$
$$\textbf{foldLeft == foldRight on monoid} \implies \quad (24)$$
$$(z :: zs + +ys).foldLeft(M.empty)(M.append) \quad (25)$$
$$\textbf{xs definition} \implies \quad (26)$$
$$(xs + +ys).foldLeft(M.empty)(M.append) \quad (27)$$

Great, we managed to prove this property with a rigorous mathematical reasoning and the best thing is we can basically write it as is in Scala and Stainless understands it!

However we are not done yet, we still need to write the parallel fold methods and ConcRope is still missing a function to generate a ConcRope from a List.

I added two parallel methods, one that is fully parallel and another one that is parallel until the trees are smaller than a given threshold. These parallel methods use a ForkJoinPool, it is the same as what we used in the *Parallelism and Concurrency* course. Now we need to prove that all these parallel methods are equivalent to one another. These proofs are not particularly interesting as they only state the 'obvious' to show that it's equivalent. What they boil down to is only saying that since computing the children in parallel is equivalent as

computing them sequentially we get the same result. They are available here[16].

Now for our case study and for practicality we need a method to convert a List to a ConcRope as it doesn't exist yet. The method itself is relatively straightforward as ConcRope already has a method to append an element.

```
def concRopeFromList[A](xs: List[A]): Conc[A] = {
  xs match {
    case Nil() => Empty[A]()
    case Cons(y, ys) => append(concRopeFromList(ys), y)
  }
} ensuring (res => res.valid &&
    res.content == xs.content &&
    res.size == xs.size &&
    res.toList == xs.reverse)
```

*Note: ConcRope has a method to prepend an element which would lead the same list, this one yields a ConcRope with reversed order, but it is slower so I added both and will use this method for the case study.*

Now, proving that this method yields the reversed list is not trivial, but it is an interesting proof. You can take a look at it in the appendix. The proof is interesting because it's segmented in different parts, first we use induction to see that:

```
xs = y :: ys

conc = concRopeFromList(xs)
conc == append(concRopeFromList(ys), y)
conc.toList == concRopeFromList(ys).toList ++ (y :: Nil)

// Induction
concRopeFromList(ys).toList == ys.reverse

conc.toList == ys.reverse ++ (y :: Nil)

// Need to show
ys.reverse ++ (y :: Nil) == ys.reverse ++ (y :: Nil).reverse
ys.reverse ++ (y :: Nil).reverse == ((y :: Nil) ++ ys).reverse
((y :: Nil) ++ ys).reverse == (y :: ys).reverse

// Namely
list1.reverse ++ list2.reverse == (list2 ++ list1).reverse
```

Then to prove this I had to prove an additional property: list concatenations are right assosciative. Actually they are assosciative but I only needed right assosciativity here. Again the code can be found in the appendix.

*Note: you can see a pattern there, most of the time proving a difficult property*

[16] https://github.com/Gorzen/stainless/blob/sorting-project/lucien/FoldMapConcRope.scala

*becomes proving many smaller properties that together prove the big property. Divide and conquer.*

# Case study: word count

Now that we have defined all this structure and the folding operations on List and ConcRope we want to know how they compare. To do this we decided to work on a word count, because it could use everything we have used so far. Here is what the code does:

1. Read a given file, creating a list of words
2. `Map` the list of words to a list of wordcount objects (`WC`), they are basically multisets; a map from `String` to `BigInt`. So, map every word to a singleton multiset.
3. *Optional step*: map the `List[WC]` → `ConcRope[WC]`
4. `Fold` (in parallel or sequentially if it's a ConcRope) the `Collection[WC]` using a `Monoid[WC]` that merges the two multisets in order to get the final `WC`.
5. From the `WC`, retrieve a `List[(String, BigInt)]`.
6. Sort the `List[(String, BigInt)]` using a `TotalOrder[(String, BigInt)]` that sorts the most used words first.
7. Write the sorted list to a file.

This case study will, thus, indeed use everything written so far and is a good method to compare List to ConcRope and to see how faster the parallel fold is to the sequential one. We then can put a file of arbitrary size to see how all these methods compare (if it fits in stack and heap).

*Note: the new Monoid and TotalOrder defined for the word count have to be verified of course.*

## Results

Here are the results, for a file of 128'457 lines, 1'095'683 words of which 81'409 unique words. It is in fact a book written in English and these are the first lines of the output:

```
the => 71744
of => 39169
and => 35968
to => 27895
a => 19811
```

### Folding

*2 core - 4 thread CPU*

| ConcRope Parallel | Time |
|---|---|
| Conversion List[WC] → ConcRope[WC] | 2.819627273 s |
| Parallel fold on ConcRope[WC], min size 32, 4 threads | 9.320218039 s |
| Parallel fold on ConcRope[WC], min size 64, 4 threads | 8.719147123 s |
| Parallel fold on ConcRope[WC], min size 32, 8 threads | 9.25718213 s |
| Parallel fold on ConcRope[WC], min size 64, 8 threads | 9.829431564 s |
| Parallel fold on ConcRope[WC], min size 32, 16 threads | 9.905632951 s |
| Parallel fold on ConcRope[WC], min size 64, 16 threads | 9.753053033 s |
| **Best total time** | **11.538774396 s** |

*Note: min size defines the threshold at which we compute the fold sequentially.*

| ConcRope Sequential | Time |
|---|---|
| Conversion List[WC] → ConcRope[WC] | 2.819627273 s |
| Sequential fold on ConcRope[WC] | 15.049214472 s |
| **Total time** | **17.868841744999997 s** |

| List | Time |
|---|---|
| Sequential foldLeft on List[WC] | 132812.152160333 s |

These results make sense, the parallel is the best especially if we fully utilize all the cores, sequential fold on ConcRope is slower and fold on List is the slowest. However, the List fold is really terrible we didn't expect it to be this bad but there is a reason for it that will be discussed further in the "*Faster multiset*" section.

### Sorting

| Sorting | Time |
|---|---|
| MergeSort on output (81'409 elements) | 0.516807123 s |
| InsertionSort on output (81'409 elements) | 82.337158522 s |

No big surprise here merge sort is significantly faster than insertion sort, even though it's on a linked list like structure. As I said earlier we used to have issues where they would have somewhat the same runtime because the splitting method was slow but as we can see it has been fixed.

## Extensions to ConcRope

As we have seen with the case study ConcRopes can be really efficient and easily parallelized, they are very interesting data structures. Thus, in the continuity of extending the library we thought it could be interesting to extend ConcRope to

make it easier to work with. Indeed, when writing code using ConcRope I realized that it is missing some functions or some syntactic sugar functions which would permit a smoother code. Our goal was to add methods to make its abstraction more list-like such that it's easier to use.

I made a list comparing the methods from List and ConcRope to see which ones are missing and started implementing the ones that seemed the most useful. I managed to add the following methods:

| Syntactic sugar | Map methods | List-like | Folding | Conversion | Predicates |
|---|---|---|---|---|---|
| `::` | `map` | `head` | `foldMap` | `toSet` | `contains` |
| `:+` | `flatMap` | `headOption` | `foldLeft` | `content` | `exists` |
| `++` | `flatten` | | `foldRight` | `fromList` | `forall` |
| `apply` | | | | `fromListReversed` | `find` |

However, while adding these methods I realized that it unfortunately isn't feasible to have an efficient and verified fully List-like abstraction with the time given. This is the case for two reasons:

1. I tried adding `reverse` and `filter` however I didn't manage to add them, it would have required much more time. The actual implementation isn't the problem, the problem was proving the postconditions we would expect from these methods. Since ConcRope has a much more complex construction as List, Stainless needs a lot more help when proving properties. Concatenation of two lists is simple you just put them side by side, but the implementation of concatenation of ConcRope is much more complex and makes it way harder to show that some properties still hold after concatenation. For example for `fold`, Stainless has trouble seeing that concatenating two filtered trees results in a filtered tree and writing the exhaustive proof is no easy task given the complex concatenation operation. For `reverse`, showing that the reversed ConcRope's list is the reverse of the original ConcRope was pretty straightforward but an exhaustive proof showing that the reversed tree is a valid one is much harder, i.e. that the tree is balanced and still holds all the properties of a ConcRope.
2. Methods such as `tail`, `drop`, `take` and all methods that get rid of a chunk of the tree are not really suited for the design of the tree. Maybe there is some way to implement some of them efficiently but the problem is if you simply remove some part of the tree, it will generally not hold some of its properties anymore. We can of course reconstruct a tree containing only the elements wanted but this is not efficient at all. Reading the many papers involving trees to find the best implementation for all of these methods would have required more time and wasn't the main focus of this project. Thus, we can only get so close to a list-like abstraction. Maybe we could find an efficient way to transform the tree in a valid way for some of these methods but this would require more time. And we would still need to prove the implementation correct, knowing that the more complex the implementation the more complex the proof usually.

## Faster multiset

As we have seen in the study case, the fold on the List took extremely long, it's supposed to be slower than ConcRope but not as slow as this. It caught our attention and we tried to understand why that was the case. It turns out that this is due to the multiset implementation of Stainless. Since we do as many unions as we have words, if we have an inefficient union operation it can drastically affect runtime. This was the former implementation of multiset, called `Bag` in Stainless:

```
case class Bag[T](theBag: scala.collection.immutable.Map[T, BigInt]) {
  def get(a: T): BigInt = theBag.getOrElse(a, BigInt(0))
  def apply(a: T): BigInt = get(a)
  def isEmpty: Boolean = theBag.isEmpty
  def +(a: T): Bag[T] =
    new Bag(theBag + (a -> (theBag.getOrElse(a, BigInt(0)) + 1)))

  def ++(that: Bag[T]): Bag[T] = new Bag[T](
    (theBag.keys ++ that.theBag.keys).toSet.map { (k: T) =>
    k -> (theBag.getOrElse(k, BigInt(0)) + that.theBag.getOrElse(k, BigInt(0)))
  }.toMap)

  def --(that: Bag[T]): Bag[T] = // Omitted
  def &(that: Bag[T]): Bag[T] = // Omitted
}
```

The method that really interests us here is `++`, it is highly inefficient. It unions all the keys of both maps, then for every key gets the value of the key in both maps and constructs a map that way. In fact, the fold on List is basically the worst case possible for this implementation and really shows the problem with it. Our fold on List is defined as a foldLeft and will proceed as such:

```
// Step 0
z: Empty - fold - w1 :: w2 :: w3 :: w4 :: w5 ...
// Step 1
z: w1 - fold - w2 :: w3 ...
//Step 2
z: (w1, w2) - fold - w3 :: w4 :: w5 ...
...
// Step ...
z: (w1,...,wn) - fold - wn+1 :: ...
```

The inefficiency comes from the fact that z becomes a huge Bag but we only union it with singleton Bags thus at every union we take all the keys of z and map them just to add one word! The bags are as unbalanced as you can get. Whereas on the ConcRope, due to the fact that the tree is balanced, you union bags that have similar sizes and thus unions are highly more efficient.

Having found the source of the problem I started implementing a more efficient Bag, keeping in mind that the bags being unioned may be very unbalanced. The former implementation is basically $O(n + m)$ where n and m are the sizes of the

bags unioned. It should be possible to achieve $O(min(n, m))$ if we only care about the smaller Bag and add it to the bigger one. The implementation I came up with is the following:

```
def ++(that: Bag[A]): Bag[A] = {
  if (that.theBag.size > theBag.size)
    // Order of a set doesn't matter
    that ++ this
  else {
    Bag(that.theBag.toSeq.foldLeft(theBag)((z, x) => {
      z.get(x._1) match {
        case None => z + ((x._1, x._2))
        case Some(i) => z.updated(x._1, x._2 + i)
      }
    }))
  }
}
```

Excellent, our union becomes a foldLeft on the smaller Bag, meaning that it now should indeed be $O(min(n, m))$!

But what about proofs? If I change the implementation of Bag, can it not break many former proofs? Fortunately, it doesn't. Bag has a special inner definition inside Stainless used when proving properties, this implementation is only used at runtime. Having an inner definition of Bag inside Stainless makes it way more efficient for proofs. This is why Stainless is so much better at proving properties involving sets; it knows many things about them that it can use to prove properties. Thus, as long as this implementation is correct we are okay, no need to change older proofs.

*Note: I also rewrote the* `--` *method using a similar approach.*

Let's see how long the fold of the case study takes now.

*4 core - 8 thread CPU, same file as previously*

| | ConcRope, 8 threads, 64 min size | List |
|---|---|---|
| List -> ConcRope | 2.819627273 s | *None* |
| Fold | 0.871780456 s | 1.12009817 s |

Very impressive results, ConcRope is still better but it's much closer. The overhead of converting the List to ConcRope has a much bigger impact now. Since the runtime is now pretty short we can see what happens on an even bigger file. To do this I had to change the case study workflow because reading the entire list of words in one go was too much for the memory, it was painfully slow. I changed it such that we compute the word count of every line while reading the file, creating one word count for every line and then we fold again on all the lines giving us the final word count. I thus also had to change the time computation, it now adds the time of every fold operation and of every conversion of List to ConcRope to

have a finer grained benchmarking result.

Here are the results for the new file, 52'957'736 words:

| | ConcRope, 8 threads, 64 min size | List |
|---|---|---|
| All conversions to ConcRope | 7.944933202 s | *None* |
| All folds | 78.286540203 s | 104.111247938 s |

I then tried to improve the Bag even further, instead of computing the proper union every time we can store the two bags when the Bags are big and that's it, giving us a tree-like structure of Bag. Then when you want to get an element you add the result of the children. This would give us a $O(1)$ union, however the time to get an element would be slower, $O(\log n)$ typically.

I tried different implementations, different ways to balance the Bags such that they are as balanced as possible and as close to the threshold size as possible; some lazy implementations, some finer grained implementations. The implementation with the best results can be found in the appendix.

Unfortunately even this one doesn't really improve the time. Plus this structure of Bags is a little unusual and makes retrieving elements slower, it can even make it ridiculously slow. Moreover, this design isn't suited for Bags subtraction which our abstraction needs to have, they can't be efficient. So I decided to come back to the simpler foldLeft implementation.

*Parallel, 8 threads, 64 min size fold, bag threshold: 10'000*

| | foldLeft union | Tree like Bags |
|---|---|---|
| All folds | 78.286540203 s | 80.284113909 s |
| Retrieving List | 0.40157084 s | 274.699970205 s |

The fold doesn't have any speedup even though it's supposed to be $O(1)$ because creating this tree is really heavy on memory. The ConcRope is huge and has to be in memory, traversing the whole thing takes a lot of time and every leaf only contains a single word, that's a lot of leaves. So if we add on top of that Bags that are trees instead of flattened Bags, it's really memory-heavy. It uses almost all 16GB of my computer.

As expected the time to retrieve list is way worse, we can see it especially well here because we retrieve a big list.

Something that could probably speed up the fold even more and would probably reduce the intensive usage of memory is having efficient arrays at leaves instead of singular elements. We would probably have better spatial locality and more efficient memory usage.

# Conclusion

I have written ordering and monoid algebraic structures in Stainless and proved their properties on some examples. I have written a case study using these structures and the methods defined on them to compare two data structures. I extended the API of the ConcRope data structure and, using the results of the case study, improved the implementation of the multiset in Stainless. I found out that extending ConcRope even more would require more time as it's difficult to translate list-like methods to a tree-like data structure that must verify certain properties.

# Further work

As we have seen, it's no easy task to extend the ConcRope even more. It would need time to think about how to implement certain methods and how to prove certain properties on the tree. However, efficient trees is a field that has already been thoroughly studied so we could find insipration from notorious researchers such as Doctor Knuth.

We could make the ConcRope more efficient by, as said earlier, having efficient mutable arrays at the leaves instead of singular elements. However, these kinds of arrays are not supported yet in Stainless so it would require adding stuff to Stainless itself.

I made the multiset more efficient, however we could do the same for other parts of the library, Set, Map, etc... We could also add some methods that are missing from the standard API to them. List has many of the standard API methods but it could be more efficient, especially regarding the stack, most of the methods are not tail recursive and some methods are really slow. However, this wouldn't be easy as we would need to update the proofs of List while having more complex methods.

# References

1. Aleksandar Prokopec and Martin Odersky. *Conc-Trees for Functional and Parallel Programming*. Raleigh, North Carolina, September 2015. `http://aleksandar-prokopec.com/resources/docs/lcpc-conc-trees.pdf`

# Appendix

## Nat

`Nat` code definition:

```
sealed abstract class Nat {
  def <(m: Nat): Boolean = {
    (this, m) match {
      case (Succ(ts), Succ(ms)) => ts < ms
      case (Zero, Succ(ms)) => true
      case _ => false
    }
  }

  def <=(m: Nat): Boolean = {
    this == m || this < m
  }
}

final case object Zero extends Nat
final case class Succ(prev: Nat) extends Nat
```

## Proof foldLeft == foldRight

```
def foldLeftEqualsFoldRight[A](xs: List[A])(implicit M: Monoid[A]): Boolean = {
 decreases(xs.size)
 (xs.foldLeft(M.empty)(M.append) == xs.foldRight(M.empty)(M.append)) because {
  xs match {
   case Nil() => {
     Nil[A]().foldLeft(M.empty)(M.append)     ==| trivial |
     M.empty                                  ==| trivial |
     Nil[A]().foldRight(M.empty)(M.append)
   }.qed
   case Cons(y, Nil()) => {
     Cons(y, Nil[A]()).foldLeft(M.empty)(M.append)     ==| trivial
     Nil[A]().foldLeft(M.append(M.empty, y))(M.append) ==| M.law_leftIdentity(y)
     Nil[A]().foldLeft(y)(M.append)                    ==| trivial
     y                                                 ==| M.law_rightIdentity(y)
     M.append(y, M.empty)                              ==| trivial
     M.append(y, Nil[A]().foldRight(M.empty)(M.append))==| trivial
     Cons(y, Nil[A]()).foldRight(M.empty)(M.append)
   }.qed
   case Cons(y1, Cons(y2, ys)) =>
     assert((y1 :: y2 :: ys).foldLeft(M.empty)(M.append) ==
       (y2 :: ys).foldLeft(M.append(M.empty, y1))(M.append))
     assert(M.law_leftIdentity(y1))
     assert((y2 :: ys).foldLeft(M.append(M.empty, y1))(M.append) ==
       (y2 :: ys).foldLeft(y1)(M.append))
     assert((y2 :: ys).foldLeft(y1)(M.append) ==
       ys.foldLeft(M.append(y1, y2))(M.append))
     assert(M.law_leftIdentity(M.append(y1, y2)))
     assert(ys.foldLeft(M.append(y1, y2))(M.append) ==
       ys.foldLeft(M.append(M.empty, M.append(y1, y2)))(M.append))
     assert(ys.foldLeft(M.append(M.empty, M.append(y1, y2)))(M.append) ==
       (M.append(y1, y2) :: ys).foldLeft(M.empty)(M.append))
     assert((M.append(y1, y2) :: ys).size < (y1 :: y2 :: ys).size)
     assert(foldLeftEqualsFoldRight(M.append(y1, y2) :: ys))
     assert((M.append(y1, y2) :: ys).foldLeft(M.empty)(M.append) ==
       (M.append(y1, y2) :: ys).foldRight(M.empty)(M.append))
     assert((M.append(y1, y2) :: ys).foldRight(M.empty)(M.append) ==
       M.append(M.append(y1, y2), ys.foldRight(M.empty)(M.append)))
     assert(M.law_associativity(y1, y2, ys.foldRight(M.empty)(M.append)))
     assert(M.append(M.append(y1, y2), ys.foldRight(M.empty)(M.append)) ==
       M.append(y1, M.append(y2, ys.foldRight(M.empty)(M.append))))
     assert(M.append(y1, M.append(y2, ys.foldRight(M.empty)(M.append))) ==
       M.append(y1, (y2 :: ys).foldRight(M.empty)(M.append)))
     assert(M.append(y1, (y2 :: ys).foldRight(M.empty)(M.append)) ==
       (y1 :: y2 :: ys).foldRight(M.empty)(M.append))
     check((y1 :: y2 :: ys).foldLeft(M.empty)(M.append) ==
       (y1 :: y2 :: ys).foldRight(M.empty)(M.append))
```

```
  }
 }
}.holds
```

## Proof ConcRopeFromList

```
def proof_concRopeFromList[A](xs: List[A]): Boolean = {
  (concRopeFromList(xs).toList == xs.reverse) because lemma_concRopeFromList(xs)
}.holds

def lemma_concRopeFromList[A](xs: List[A]): Boolean = {
  decreases(xs.size)
  xs match {
    case Nil() =>
      assert(concRopeFromList(Nil[A]()).toList == Nil[A]())
      assert(Nil[A]() == Nil[A]().reverse)
      check(concRopeFromList(Nil[A]()).toList == Nil[A]().reverse)
    case Cons(y, ys) => {
      assert(concRopeFromList(y :: ys).toList ==
            append(concRopeFromList(ys), y).toList)
      assert(append(concRopeFromList(ys), y).toList ==
            concRopeFromList(ys).toList ++ Cons(y, Nil[A]()))
      assert(lemma_concRopeFromList(ys))
      assert(concRopeFromList(ys).toList ++ Cons(y, Nil[A]()) ==
            ys.reverse ++ Cons(y, Nil[A]()))
      assert(ys.reverse ++ Cons(y, Nil[A]()) ==
            ys.reverse ++ Cons(y, Nil[A]()).reverse)
      assert(concat_reverse(Cons(y, Nil[A]()), ys))
      assert(ys.reverse ++ Cons(y, Nil[A]()).reverse ==
            (Cons(y, Nil[A]()) ++ ys).reverse)
      assert((Cons(y, Nil[A]()) ++ ys).reverse == (y :: ys).reverse)
      check(concRopeFromList(xs).toList == xs.reverse)
    }
  }
}
```

```
def concat_reverse[A](xs: List[A], ys: List[A]): Boolean = {
  decreases(xs.size)
  ((xs ++ ys).reverse == ys.reverse ++ xs.reverse) because {
    xs match {
      case Nil() => {
        (Nil[A]() ++ ys).reverse ==| trivial |
        ys.reverse ==| trivial |
        ys.reverse ++ Nil[A]() ==| trivial |
        ys.reverse ++ Nil[A]().reverse
      }.qed
      case Cons(z, zs) => {
        ((z :: zs) ++ ys).reverse ==| trivial |
        (z :: zs ++ ys).reverse ==| trivial |
        (zs ++ ys).reverse :+ z ==| concat_reverse(zs, ys) |
        (ys.reverse ++ zs.reverse) :+ z ==| trivial |
        ys.reverse ++ zs.reverse :+ z
                ==| right_assosciative(ys.reverse, zs.reverse, z) |
        ys.reverse ++ (zs.reverse :+ z) ==| trivial |
        ys.reverse ++ (z :: zs).reverse
      }.qed
    }
  }
}.holds

def right_assosciative[A](xs: List[A], ys: List[A], y: A): Boolean = {
  decreases(xs.size)
  (xs ++ ys :+ y == xs ++ (ys :+ y)) because {
    xs match {
      case Nil() => {
        Nil[A]() ++ ys :+ y ==| trivial |
        ys :+ y ==| trivial |
        Nil[A]() ++ (ys :+ y)
      }.qed
      case Cons(z, zs) => {
        (z :: zs) ++ ys :+ y ==| trivial |
        z :: (zs ++ ys :+ y) ==| right_assosciative(zs, ys, y) |
        z :: (zs ++ (ys :+ y)) ==| trivial |
        z :: zs ++ (ys :+ y) ==| trivial |
        (z :: zs) ++ (ys :+ y)
      }.qed
    }
  }
}.holds
```

## Bag union, SmallBag - BigBag

```
def ++(that: Bag[A]): Bag[A] = {
  if (that.size > this.size)
    that ++ this
  else (this, that) match {
    case (SmallBag(theMap), SmallBag(thatMap)) if theMap.size <= threshold =>
        // concatMaps is equivalent to the foldLeft union we saw before
        SmallBag(concatMaps(theMap, thatMap))
    case (SmallBag(theMap), SmallBag(thatMap)) => BigBag(this, that)
    case (SmallBag(theMap), BigBag(left, right)) if theMap.size <= threshold =>
        smallBagConcat(SmallBag(theMap), that.flatten)
    case (SmallBag(theMap), BigBag(left, right)) => BigBag(this, that)
    case (BigBag(left, right), SmallBag(thatMap)) if right.size <= threshold =>
        BigBag(left, right ++ SmallBag(thatMap))
    case (BigBag(left, right), SmallBag(thatMap)) => BigBag(this, that)
    case (BigBag(l1, r1), BigBag(l2, r2)) if r1.size <= threshold =>
        BigBag(BigBag(l1, r1 ++ r2), l2)
    case (BigBag(l1, r1), BigBag(l2, r2)) => BigBag(this, that)
    }
}
```