Advancing Algebraic Reasoning for Scala

Lucien Michaël Iseli

June 15th, 2019

Project

Stainless

- Proves properties statically
- Supports a good chunk of functional Scala

Goal

- Write a bigger verified project using Stainless
- Algebraic structures
- Proofs
- Real world results

Algebraic structures - Comparison

Equality

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

Partial Order - defined with $\leq$

- Reflexive: $\forall x : x \leq x$
- Antisymmetric: $\forall x, y : x \leq y \land y \leq x \implies x = y$
- Transitive: $\forall x, y, z : x \leq y \land y \leq z \implies x \leq z$

Total Order - defined with $\leq$

- Connex: $\forall x, y : x \leq y \lor y \leq x$

Code Equality

```scala
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)
    }
}
```

Example: Total order

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

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

Sorting!

- Insertion sort
- Merge sort

Algebraic Structures - Monoid

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

A monoid is a semigroup but it has additional properties.

Semigroup

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

Monoid

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

Algebraic Structures - Monoid - Code

```
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)
    }
}
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
    }
}
```

Fold - List

Fold is an important method and is interesting when defined on monoids

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

- Proved simple cases, i.e. sum is correct
- Interesting: `foldLeft == foldRight` thanks to monoids (omitting base cases, `xs.length >= 2`)
  - Stainless better with foldRight

Proof - foldLeft == foldRight

$$xs.foldLeft(M.empty)(M.append)$$
$$(y1 :: y2 :: ys).foldLeft(M.empty)(M.append)$$
$$(y2 :: ys).foldLeft(M.append(M.empty, y1))(M.append)$$
$$(y2 :: ys).foldLeft(y1)(M.append)$$
$$ys.foldLeft(M.append(y1, y2))(M.append)$$
$$ys.foldLeft(M.append(M.empty, M.append(y1, y2)))(M.append)$$
$$(M.append(y1, y2) :: ys).foldLeft(M.empty)(M.append)$$
$$(M.append(y1, y2) :: ys).foldRight(M.empty)(M.append)$$
$$M.append(M.append(y1, y2), ys.foldRight(M.empty)(M.append)$$
$$M.append(y1, M.append(y2, ys.foldRight(M.empty)(M.append)$$
$$M.append(y1, (y2 :: ys).foldRight(M.empty)(M.append))$$
$$(y1 :: y2 :: ys).foldRight(M.empty)(M.append)$$
$$xs.foldRight(M.empty)(M.append)$$

Fold - ConcRope

Balanced, easily parallelized tree-like data structure.

Complex union.

Paper by Aleksandar Prokopec.

Fold - ConcRope

```scala
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]

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))
  }
}
```

Fold - ConcRope

- Interesting proof: `xs: Conc[A] - fold(xs.toList) == fold(xs)`
  - Correct implementation
  - `fold(xs)` → `fold(xs.toList)` → `foldRight(xs.toList)`
- Parallel folds, proofs

`List[A]` → `Conc[A]`

- Conc has append and prepend methods
- `fromList` with prepend (faster) ⟹ reverse list
- Proof `fromList(xs).toList == xs.reverse`
  - `(xs ++ ys).reverse == ys.reverse ++ xs.reverse`
  - `xs ++ ys ++ zs == xs ++ (ys ++ zs)` - right assossiative

Case study: word count

1. Read file $\implies$ `List[String]`, list of words.
2. Map `List[String]` $\rightarrow$ `List[WC]`. *WC are multisets of* `String`.
3. *Optional step*: map the `List[WC]` $\rightarrow$ `ConcRope[WC]`.
4. `Fold` (parallel or sequential) the `Collection[WC]` with `Monoid[WC]` $\implies$ get final `WC`.
5. From the `WC`, retrieve a `List[(String, BigInt)]`.
6. Sort the `List[(String, BigInt)]` using a `TotalOrder[(String, BigInt)]`.
7. Write the sorted list to a file.

Case study: word count - Results

File of 1'095'683 words, containing 81'409 unique words

The first five lines of the output

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

Case study: word count - Results

2 core - 4 thread CPU

ConcRope Parallel	Time
Conversion List[WC] → ConcRope[WC]	2.819627273s
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**

Case study: word count - Results

ConcRope Sequential	Time
Conversion List[WC] $\rightarrow$ 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

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

Extensions to ConcRope

Syntactic sugar	Map methods	List-like
`::`	`map`	`head`
`:+`	`flatMap`	`headOption`
`++`	`flatten`	
`apply`		

Conversion	Folding	Predicates
`toSet`	`foldMap`	`contains`
`content`	`foldLeft`	`exists`
`fromList`	`foldRight`	`forall`
`fromListReversed`		`find`

Extensions to ConcRope

- Adding other methods is no easy task
- They need to be efficient on balanced trees
- Proofs of correctness

Faster multiset

Terribly slow fold on List.

Inefficient multiset in library, union $O(n + m)$

```
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)
```

Faster multiset

Why is it so slow on List but not on ConcRope?

`z: Empty - Bag(w1) :: Bag(w2) :: Bag(w3) :: Bag(w4) :: ...`

Faster multiset

Why is it so slow on List but not on ConcRope?

```
z: Bag(w1)  -  Bag(w2) :: Bag(w3) :: Bag(w4) :: ...
```

Faster multiset

Why is it so slow on List but not on ConcRope?

```
z: Bag(w1, w2)  -  Bag(w3) :: Bag(w4) :: ...
```

Faster multiset

Why is it so slow on List but not on ConcRope?

```
z: Bag(w1, w2, w3)  -  Bag(w4) :: ...
```

Faster multiset

Why is it so slow on List but not on ConcRope?

...

```
z: Bag(w1, ..., wn)  -  Bag(wn+1) :: Bag(wn+2) :: ...
```

$O(m + n)$

The Bags are as unbalanced as possible, even though we only add one word the operation is really slow. It doesn't get a lot of work done.

ConcRopes will result in balanced unions by construction so they are more efficient.

This is basically the worst case for this union.

Faster multiset

$O(min(n, m))$ should be possible

```scala
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)
            }
        }))
    }
}
```

Faster multiset - Results

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

With a much bigger file (52'957'736 words), changed main such that it supports bigger files (folds every line).

	ConcRope, 8 threads, 64 min size	List
Conv. to ConcRope	7.944933202 s	*None*
All folds	78.286540203 s	104.111247938 s

Faster multiset

Tried to go further using amortized $O(1)$ append with a tree-like data structure.

Wasn't successful because of the huge memory cost + cost of traversing the whole ConcRope

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

Plus, retrieving the list is much worse by construction.

Further work

- Efficient mutable arrays at leaves of ConcRope for better memory utilization and faster execution
- Extend ConcRope even more, efficient and proved
- Improve and speedup other parts of the Stainless library