6 November 2018

Anatomy of semigroups and monoids

by Myself

I will try to group here, in an anatomy atlas, basic notions of functional programming that I find myself explaining often lately into a series of articles.

The idea here is to have a place to point people needing explanations and to increase my own understanding of these subjects by trying to explain them the best I can. I'll try to focus more on making the reader feel an intuition, a feeling about the concepts rather than on the perfect, strict correctness of my explanations.

- Part 1: Anatomy of functional programming
- Part 2: Anatomy of an algebra
- Part 3: Anatomy of a type class
- Part 4: Anatomy of semi groups and monoids
- Part 5: Anatomy of functors and category theory
- Part 6: Anatomy of the tagless final encoding - Yet to come !

What is a *semigroup* ?

General definition

Semigroup (and *monoid*, you'll see later) is a complicated word for a **really** simple concept. We'll cover quickly *semigroups* and we'll explain longer *monoids* since they are strongly related.

Wikipedia's definition is:

> *In mathematics, a semigroup is an algebraic structure consisting of a set together with an associative binary operation.*

Ok, that sounds a bit abstract, let's try to re-phrase it with programming terms:

In the context of programming, a *semigroup* is composed of two things:

1. A type `A`
2. An associative operation combining two values of type `A` into a value of type `A`, let's call it `combine`
    - That would be a *function* with a type signature: `(A, A) => A`
    - Which is associative, meaning that the order in which you combine elements together (where you decide to put your parenthesis) does not matter
        - `combine(combine(a1, a2), a3) == combine(a1, combine(a2, a3))` with `a1`, `a2`, `a3` values of type `A`

Then it is said that `A` **forms a *semigroup* under** `combine`.

Some examples

Integer under addition

- type: `Int`
- operation: `+`

Indeed,

- + type here is: `(Int, Int) => Int`
- + is associative `(20 + 20) + 2 == 20 + (20 + 2)`

Integers form a *semigroup* under addition.

Boolean under OR

- type: `Boolean`
- operation: ||

Indeed,

- || type here is: `(Boolean, Boolean) => Boolean`
- || is associative `(true || false) || true == true || (false || true)`

Booleans form a *semigroup* under OR.

List under list concatenation

- type: `List[A]`
- operation: ++

Indeed,

- ++ type here is: `(List[A], List[A]) => List[A]`
- ++ is associative `(List(1, 2) ++ List(3, 4)) ++ List(5, 6) == List(1, 2) ++ (List(3, 4) ++ List(5, 6))`

More examples !

- Integers under multiplication
- Booleans under AND
- String under concatenation
- A LOT more.

We'll now explore *monoids* since they are a "upgraded" version of *semigroups*.

What is a *monoid* ?

General definition

Given the definition of a *semigroup*, the definition of a *monoid* is pretty straight forward:

> *In mathematics, a monoid is an algebraic structure consisting of a set together with an associative binary operation and an identity element.*

Which means that a *monoid* is a *semigroup* plus an identity element.

In our programming terms:

In the context of programming, a *monoid* is composed of two things:

1. A *semigroup*:
    - A type `A`
    - An associative operation combining two values of type `A` into a value of type `A`, let's call it `combine`

2. An identity element of type `A`, let's call it `id`, that has to obey the following laws:
    - `combine(a, id) == a` with a a value of type `A`
    - `combine(id, a) == a` with a a value of type `A`

Then it is said that `A` **forms a *monoid* under** `combine` **with identity element** `id`.

Some examples

We could take our *semigroups* examples here and add their respective identity elements:

- Integer under addition
    - With identity element 0:
        - `42 + 0 = 42`
        - `0 + 42 = 42`

- Boolean under OR
    - With identity element `false`:
        - `true || false == true`, `false || true == true`
        - `false || false == false`

- List under list concatenation

- With identity element `Nil` (empty List):
  - `List(1, 2, 3) ++ Nil == List(1, 2, 3)`
  - `Nil ++ List(1, 2, 3) == List(1, 2, 3)`

Whenever you have an identity element for your *semigroup*'s type and `combine` operation that holds the identity laws, then you have a *monoid* for it.

But be careful, there are some *semigroups* which are not *monoids*:

Tuples form a *semigroup* under first (which gives back the tuple's first element).

- type: `Tuple2[A, A]`
- operation: `first` (`def first[A](t: Tuple2[A, A]): A = t._1`)

Indeed,

- `first` type here is: `Tuple2[A, A] => A`
- `first` is associative `first(Tuple2(first(Tuple2(a1, a2)), a3)) == first(Tuple2(a1, first(Tuple2(a2, a3))))` with a1, a2, a3 values of type `A`

But there is no way to provide an identity element `id` of type `A` so that:

- `first(Tuple2(id, a)) == a` and `first(Tuple2(a, id)) == a` with a a value of type `A`

What the hell is it for ?

Monoid is a functional programming constructs that **embodies the notion of combining "things" together**, often in order to reduce "things" into one "thing". Given that the combining operation is associative, it can be **parallelized**.

And that's a **BIG** deal.

As a simple illustration, this is what you can do, absolutely fearlessly when you know your type `A` forms a *monoid* under `combine` with identity `id`:

- You have a huge, large, massive list of `A`s that you want to reduce into a single `A`
- You have a cluster of N nodes and a master node

- You split your huge, large, massive list of `A`s in N sub lists
- You distribute each sub list to a node of your cluster
- Each node reduce its own sub list by `combining` its elements 2 by 2 down to 1 final element
- They send back their results to the master node
- The master node only has N intermediary results to `combine` down (in the same order as the sub lists these intermediary results were produced from, remember, associativity !) to a final result

You successfully, without any fear of messing things up, parallelized, almost for free, a reduction process on a huge list thanks to *monoids*.

Does it sound familiar ? That's naively how fork-join operations works on Spark ! Thank you *monoids* !

How can we encode them in Scala ?

Semigroups and *monoids* are encoded as type classes.

We are gonna go through a simple implementation example, you should never have to do it by hand like that since everything we'll do is provided by awesome FP libraries like Cats or Scalaz.

Here are our two type classes:

```
trait Semigroup[S] {
    def combine(s1: S, s2: S): S
}

trait Monoid[M] extends Semigroup[M] {
    val id: M
}
```

And here is my business domain modeling:

```
type ItemId = Int
case class Sale(items: List[ItemId], totalPrice: Double)
```

I want to be able to combine all my year's sales into one big, consolidated, sale.

Let's define a *monoid* type class instance for `Sale` by defining:

- `id` being an empty `Sale` which contains no item ids, and 0 as `totalPrice`
- `combine` as concatenation of item id lists and addition of `totalPrices`

```
implicit val saleMonoid: Monoid[Sale] = new Monoid[Sale] {
    override val id: Sale                                  = Sale(List.
    override def combine(s1: Sale, s2: Sale): Sale = Sale(s1.i
}
```

Then I can use a lot of existing tooling, generic functions, leveraging the fact that the types they are working on are instances of *monoid*.

`combineAll` (which is also provided by *Cats* or *Scalaz*) is one of them and permit to, generically, combine all my sales together for free !

```
def combineAll[A](as: List[A])(implicit M: Monoid[A]): A = {
    def accumulate(accumulator: A, remaining: List[A]): A = rem
        case Nil          ⇒ accumulator
        case head :: tail ⇒ accumulate(M.combine(accumulator, h
    }
    accumulate(M.id, as)
}

val sales2018: List[Sale] = List(Sale(List(0), 32), Sale(List(
val totalSale: Sale       = combineAll(sales2018) // Sale(List(
```

Nota bene: Here, for sake of simplicity, I did not implement `combineAll` with `foldLeft` so I don't have to explain `foldLeft`, but you should know that my `accumulate` inner function **is** `foldLeft` and that `combineAll` should in fact be implemented like that:

```
def combineAll[A](as: List[A])(implicit M: Monoid[A]): A = as.
```

Voilà !

More material

If you want to keep diving deeper, some interesting stuff can be found on my FP resources list and in particular:

- Scala with Cats - Semigroup and monoid chapters
- Why Spark can't foldLeft: Monoids and Associativity
- Cats documentation
- Let me know if you need more

Conclusion

To sum up, we saw:

- How simple **semigroups** and **monoids** are and how closely related they are
- We saw examples of *semigroups* and *monoids*
- We had an insight about how useful these FP constructs can be in real life
- And finally we showed how they are encoded in *Scala* and had a glimpse on what you can do with it thanks to major FP librairies

I'll try to keep that blog post updated. If there are any additions, imprecision or mistakes that I should correct or if you need more explanations, feel free to contact me on Twitter or by mail !

tags: *Scala* - *Functional programming*

Hosted on GitHub Pages — Theme by orderedlist