5 — Algebras, Laws, Monoids

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2020

Lehrstuhl für Informatik VI, Uni Würzburg

Algebras and Laws

Algebras

- Often following a signature's types to an implementation is possible.
- Sometimes, concrete domain not even relevant, e.g. implementing `map` in terms of `flatMap` and a single parameter constructor.
- Similar reasoning to simplifying an algebraic equation

$\Rightarrow$ Treating API as *algebras*

- Algebra in mathematical sense: one or more sets, functions operating on these sets, axioms
- In our case: sets are types like `Option[A]` or `List[Option[A]]`, functions are operations like `map`, `fold`, ...

Laws

- Let's formalize our reasoning about our APIs.
- Start with some law that seems reasonable
- Example: law of mapping over lists:

```
List(1).map(_ + 1) == List(2)
```

- Might be used as a test case for our List implementation

Generalizing Laws

Not very useful in that form, generalize it

```
List(x).map(f) == List(f(x))   // Remove constants, insert variable function
List(x).map(id) == List(id(x)) // Substitute identity for f
List(x).map(id) == List(x)     // Simplify
y.map(id) == y                 // Generalize for lists of any length
```

- Law now only talks about `map`

Law of mapping over lists

```
y.map(id) == y
```

What does our law say about `map`?

- It *can't* throw an exception before applying the function
- It leaves `y` unaffected, if the given function is identity
- We can also substitute to get back more specific law from beginning

Why have laws about code and proofs

In ordinary programming, stating laws and proving properties is uncommon. Why is it important in functional programming?

- In FP: expected to factor out functionality into *composable* components
- Already seen: side effects hurt compositionality
- More generally: any hidden assumptions / behaviour, that prevent treating component as black box make composition difficult
- Giving API an algebra with laws allows to treat API objects as black boxes

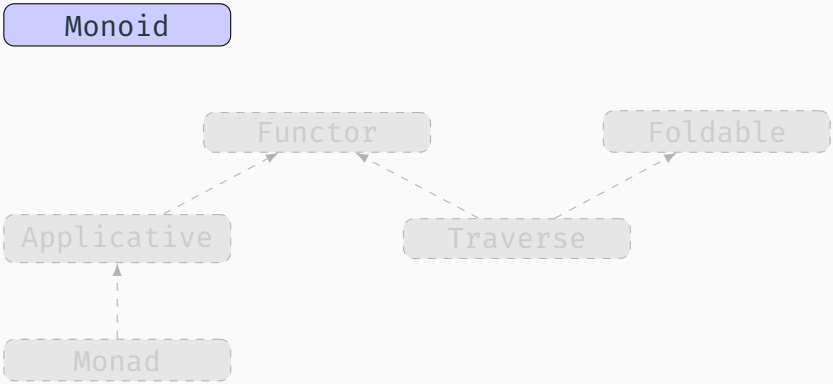
Further Reading

Chapters 7 – 9 of the red book [1]. You can find practical examples on algebraic API design in these chapters.

[1] Functional Programming in Scala by Chiusano, Bjarnason

Monoids

Typeclass hierarchy



Purely algebraic structures

- Monoids are purely algebraic structures, i.e. only defined by their algebras.
- More such structures in the next lectures. Monoids are pretty simple, ubiquitous and useful, so we'll start with them.

What is a monoid?

Let's look at some algebras, starting with string concatenation:

- `"foo" + "bar"` gives us `"foobar"`
- Concatenating with `""` changes nothing, i.e. it's the identity element.
- For three strings `r`,`s`,`t`, operation `(r + s + t)` is associative, i.e. these are equivalent:
  - `(r + (s + t))`
  - `((r + s) + t)`

What is a monoid?

The same rules apply to more algebras:

- Integer addition: `+` is associative, `0` is identity
- Integer multiplication: `*` is associative, `1` is identity
- Boolean `&&` and `||` are associative, with identities *`true`* resp. *`false`*.

Monoids

- Algebras like these are very common. They are called Monoids.
- Laws of associativity and identity collectively called monoid laws
- A Monoid consists of:
  - Some type `A`
  - Associative operation `combine` that combines two values of type `A` into one.
    `combine(combine(x, y), z) == combine(x, combine(y, z))`
    for any `x: A, y: A, z: A`
  - A value `zero: A`, such that
    `combine(zero, x) == x` and `combine(x, zero) == x`
    for any `x: A`

A Monoid trait

In Scala, we define this as a trait:

```
trait Monoid[A]:
  /** must be associative */
  def combine(a1: A, a2: A): A

  /** must be identity element */
  def zero: A
```

Monoid trait: examples

```
trait Monoid[A]:
  def combine(a1: A, a2: A): A
  def zero: A
```

Implementation for string concatenation:

```
def stringMonoid: Monoid[String] = new Monoid:
  def combine(a1: String, a2: String) = a1 + a2
  def zero = ""
```

Implementation for list concatenation:

```
def listMonoid[A]: Monoid[List[A]] = new Monoid:
  def combine(a1: List[A], a2: List[A]) = a1 ++ a2
  def zero = Nil
```

Exercise: Basic monoids

Give the following monoid instances (all very similar to the string monoid):

```
def intAddition: Monoid[Int]
def intMultiplication: Monoid[Int]
def booleanOr: Monoid[Boolean]
def booleanAnd: Monoid[Boolean]
```

Exercise: Basic monoids — solution

```
def intAddition: Monoid[Int] = new Monoid:
  def combine(a1: Int, a2:Int) = a1 + a2
  def zero = 0

def intMultiplication: Monoid[Int] = new Monoid:
  def combine(a1: Int, a2:Int) = a1 * a2
  def zero = 1

def booleanOr:  Monoid[Boolean] = new Monoid:
  def combine(a1: Boolean, a2:Boolean) = a1 || a2
  def zero = false

def booleanAnd: Monoid[Boolean] = new Monoid:
  def combine(a1: Boolean, a2:Boolean) = a1 && a2
  def zero = true
```

Exercise: Monoid for Option

Give a monoid instance for `Option` values:

```
def optionMonoid[A]: Monoid[Option[A]]
```

Hints:

- Because of the abstract definition in parameter A, number of possible implementations is limited. But more than one implementation satisfies laws.
- When using only methods we mentioned in the lecture, your implementation of `combine` may be longer than the previous ones.

Exercise: Monoid for Option — Solution

```
def optionMonoid[A]: Monoid[Option[A]] = new Monoid:
  def combine(x: Option[A], y: Option[A]) = x match
    case Some(_) => x  // first operand is present, return it
    case None => y     // otherwise return second operand
  def zero = None
```

Gives us first option if it's a `Some`, else gives second option. Reversed implementation is also correct.

Using standard library, this is equivalent to `orElse`:

```
def optionMonoidStd[A]: Monoid[Option[A]] = new Monoid:
  def combine(x: Option[A], y: Option[A]) = x orElse y
  def zero = None
```

Exercise: Monoid for Option — Solution

Why would always returning `None` not be a valid Monoid?

Exercise: Monoid for Option — Solution

Why would always returning `None` not be a valid Monoid?

Because then our zero value would no longer fulfill the identity law (`combine(zero, x) == x` and `combine(x, zero) == x` for any `x`). If we pass a `Some` and `zero`, the law requires us to return the same `Some`.

Duals of monoids

- As seen, sometimes two implementations possible
- The monoid with inversed operand order is called the dual of a monoid.
- If the operand is also commutative, the monoid is equivalent to its dual, e.g. the integer and boolean monoids defined previously.
- To get the dual, simply swap operands:

```
def dual[A](m: Monoid[A]): Monoid[A] = new Monoid[A]:
  def combine(x: A, y: A): A = m.combine(y, x)
  val zero = m.zero
```

Exercise: Monoid for endofunctions

Functions with same argument and return type are also called endofunctions.
Write a monoid for such functions:

```
def endoMonoid[A]: Monoid[A => A]
```

Exercise: Monoid for endofunctions — Solution

```
def endoMonoid[A]: Monoid[A => A] = new Monoid:
  def combine(f: A => A, g: A => A) = a => f(g(a))  // f.compose(g)
  def zero = (a: A) => a
```

Alternatively, the dual:

```
def combine(f: A => A, g: A => A) = a => g(f(a))  // f.andThen(g), rest identical
```

Typeclasses

`Monoid` is an example of a typeclass.

Typeclasses can be used for *polymorphism*, as an alternative to inheritance. We'll take look at the differences between typeclass and inheritance polymorphism.

Then we will see how Scala makes typeclasses nicer to use with some new syntax.

Polymorphism

Forms of Polymorphism

The goal of polymorphism is to write code in a more reusable, generic fashion, i.e. make it working with different types. There are three main forms of polymorphism:

Subtype Polymorphism: using inheritance to have code work with subtypes

Parametric Polymorphism: using generics, which replace types by type variables

Ad-hoc Polymorphism: using typeclasses to define a common interface for independent types

Polymorphism via inheritance

When we write polymorphic code in typical OOP languages, we define an interface and classes inheriting this interface, implementing it.

```
trait Shape:
  def area: Double

case class Circle(r: Double) extends Shape:
  def area: Double = math.Pi * r * r

case class Rectangle(width: Double, length: Double) extends Shape:
  def area: Double = width * length
```

Polymorphism via inheritance

When we want to use this in a method, we define the parameter with the interface type. We constrain our method to accept only classes that implement Shape:

```
def areaOf(shape:Shape) = shape.area

areaOf(Circle(2.0))
areaOf(Rectangle(4.2, 2.3))
```

The runtime will choose the right implementation of area based on the passed parameter.

Polymorphism via typeclasses

Typeclasses are used to place constraints on type variables. A function with a parametric type `A` may for example require an additional parameter of type `Monoid[A]`, which restricts `A` to types for which such an instance exists. For example, we can use our monoids to combine all elements in a list:

```
def combineAll[A](as: List[A])(m: Monoid[A]): A =
  as.foldLeft(m.zero)(m.combine)
```

Therefore, typeclasses in Scala are encoded as traits with at least one type parameter.

Switching to typeclasses

We can implement our shape example from before using typeclasses instead of inheritance:

```
//data classes defined independently of Shape trait
case class Circle(r: Double)
case class Rectangle(width: Double, length: Double)

trait Shape[A]:
  def area(a: A): Double

val circleShape = new Shape[Circle]:
  def area(c:Circle): Double = math.Pi * c.r * c.r

val rectangleShape = new Shape[Rectangle]:
  def area(rect: Rectangle): Double = rect.width * rect.length
```

Using typeclasses

Our `areaOf` method now looks a bit more complex:

```
def areaOf[A](shape:A)(S: Shape[A]) = S.area(shape)
```

Our methods are no longer on the same object as the data. And calling it requires more parameters:

```
areaOf(Circle(2.0))(circleShape)
```

We will see how Scala lets us write this in a nicer way later.

When to use typeclasses?

You may ask yourself, why to use typeclasses, if using traits as interfaces seem simpler. But typeclasses can express things that interfaces cannot.

We will look at the differences more closely now and give guidelines, when to use one or the other.

Typeclasses: constructors

Typeclasses can contain functions that do not require an instance of the type, like constructors. This is not possible with an OOP-style interface.

For example, the string concatenation monoid (a `Monoid[String]`) contains `zero`, which returns an empty string `""` without needing parameters. If you used an interface here, you would need a `String`, on which to call the method.

Typeclasses: definition

You can define a typeclass instance for any type (as long as the methods can be implemented). You do not need to change the existing type.

We defined monoids for various types from the standard library, while we wouldn't have been able to do that with interfaces (we can't have a class inherit something without changing it).

Typeclasses: conditional instances

Typeclasses can have instances, that can be generated when certain conditions are met. We will later see examples of such monoids, e.g. one for 2-tuples, which is only defined if both types in the tuple also have a monoid.

If we modeled monoids with inheritance, such a construction wouldn't be possible.

Typeclasses: multiple variants

Scala does not enforce *global uniqueness* of typeclasses, so we can have several instances of the same typeclass for the same type. We've already seen two `Monoid[Int]`. In contrast, a class cannot implement the same interface with the same types twice.

BUT: usually there should be only one instance of a typeclass for a given type. Aside from reducing confusion, this allows us to have the compiler look them up for us. We'll see how in a few slides.

Interfaces: storing

One situation, where an inherited trait is more suited, is when you want collections of the common supertype. With the typeclass for shapes, it is not possible (in Scala) to have a `List[Shape]`.

So our **Shape** example is actually one of the situations, where you would rather use an inherited trait.

Note that we *can* write something like `List[T]` and force `T` to have a shape instance, but this is different, as it does not allow different shapes in the same list.

Example: a typeclass in Java

There's actually a great example of typeclasses vs. inheritance in the Java standard library: `Comparable` and `Comparator`.

Comparable inheritable interface. Class can only have a single implementation per type it can be compared to. Does not require additional parameters in functions.

Comparator like a typeclass, external to the class. Can be defined in several ways for a type according to the required ordering. Using `comparing` method, can derive a comparator based on a field that has itself a comparator. Always has to be passed around explicitly.

As both have pros and cons, both are regularly used and sometimes combined.

When to use typeclasses

So overall, you should use a typeclass, when:

- the possible instances of your typeclass are unrelated, i.e. it makes no sense for them to have a common supertype.
- lots of the implementation is type-specific
- you want to be able to add instances for existing types

You should use inheritance, when:

- you want to group elements inheriting from the same supertype together in a collection

They are also not mutually exclusive, a type can inherit traits and have typeclass instances at the same time.

Givens and Using

Monoids and lists

Recall the definitions of folds on `List[A]`

```
def foldRight[B](z: B)(f: (A, B) => B): B
def foldLeft[B](z: B)(f: (B, A) => B): B
```

What do we get, when A and B are the same type?

```
def foldRight(z: A)(f: (A, A) => A): A
def foldLeft(z: A)(f: (A, A) => A): A
```

These types match the parts of our `Monoid[A]` definition!

Reducing boilerplate with given instances

We've already seen this can be used to generalize list element concatenation:

```
def combineAll[A](as: List[A])(m: Monoid[A]): A =
  as.foldLeft(m.zero)(m.combine)
```

As typeclasses, in contrast to interfaces, are implemented separately from the types that fulfill their requirements, we need to pass them around all the time. Ideally, we'd like to have the compiler know from context, what instance to use.

If we call `combineAll` with a list of strings, we need a monoid for strings. Now if we limit us to defining only unique instances, i.e. never define multiple instances of a typeclass for the same type, there is only a single valid value we can pass for the monoid parameter.

Givens

Givens are a concept in Scala for defining “canonical” values for a type, for example for a typeclass, but also for other types that have to be passed around a lot.

A method parameter can be marked with the keyword `using`, to have the compiler look up such a canonical value and fill it in. The lookup is only based on the *type*, so having two given values in scope with the same type will cause an error on lookup.

Givens — example

```
// define a type that will pe passed around
case class MyContext(x: Int)

// define a canonical instance. Name is optional, as lookup is by type
given primaryContext: MyContext = MyContext(2)

// require a given value from scope by marking a parameter list with `using`
def useContext(y: Int)(using ctx: MyContext) = ctx.x + y

// call the function without its `using` parameter list
// the compiler looks up the parameter from given values in scope and adds them
useContext(3) // = 2 + 3 = 5

// explicitly specifying `using` is allowed for overriding the given value
useContext(3)(using MyContext(5))
```

Givens for typeclasses

In case of typeclasses, our instances should be unique, so we usually don't name them. Also, the given values are usually anonymous classes that implement a trait. To support this use of givens, Scala provides a shorter syntax to define a typeclass instance.

Lets convert some of our instances for Monoid:

Normal value
with anonymous class

```
def stringMonoid: Monoid[String] = new Monoid:
  def combine(a1: String, a2: String) = a1 + a2
  def zero = ""
```

Given value
typeclass syntax

```
given Monoid[String] with
  def combine(a1: String, a2: String) = a1 + a2
  def zero = ""
```

Using givens

We can modify our `combineAll` now to use a given instance:

```
def combineAll[A](as: List[A])(using m: Monoid[A]): A =
  as.foldLeft(m.zero)(m.combine)
```

The compiler will look up a matching monoid instance, when we call `combineAll` with a concrete type. If the type is a parameter at the call site too, the calling function also needs a `using` clause. When only passing the instance on to other functions, it does not need a name.

```
def filteredCombine[A](as: List[A])(f: A => Boolean)(using Monoid[A]): A =
  combineAll(as.filter(f))
```

Extension methods

If we want our type with a typeclass instance to have methods, that look like they are defined on our type, we can use *extension methods*:

```
trait Monoid[A]:
  def combine(a1: A, a2: A): A
  def zero: A

  /** allows us to write `a1 |+| a2` instead of `combine(a1, a2)` */
  extension (a1: A)
    def |+| (a2: A) = combine(a1, a2)
```

This allows using the method `|+|` for any type that has a given `Monoid` in scope.

```
def addExample[A](first: A, second: A)(using Monoid[A]): A =
  first |+| second
```

Monoids compose

- Monoids can be composed.
- If `A` and `B` are monoids, then tuple type `(A, B)` is also monoid (called their *product type*).
- Can be implemented generically:

```
given product[A,B](using MA: Monoid[A], MB: Monoid[B]) : Monoid[(A, B)] with
  def combine(x: (A, B), y: (A, B)) =
    (x._1 |+| y._1, x._2 |+| y._2)

  def zero = (MA.zero, MB.zero)
```

Monoid composition

- Some type constructors can form monoids, if their parameters are monoids
- Example: monoid that merges maps, whose values are a monoid:

```
given mapMergeMonoid[K,V](using MV: Monoid[V]): Monoid[Map[K, V]] with
  def zero = Map[K,V]()
  def combine(a: Map[K, V], b: Map[K, V]) =
    (a.keySet ++ b.keySet).foldLeft(zero) ( (acc,k) =>
      acc + (
        k -> (a.getOrElse(k, MV.zero) |+| b.getOrElse(k, MV.zero))
      )
    )
```

Monoid composition

- Some type constructors can form monoids, if their parameters are monoids
- Example: monoid that merges maps, whose values are a monoid:

```
given mapMergeMonoid[K,V](using MV: Monoid[V]): Monoid[Map[K, V]] with
  def zero = Map[K,V]()
  def combine(a: Map[K, V], b: Map[K, V]) =
    (a.keySet ++ b.keySet).foldLeft(zero) ( (acc,k) =>
      acc + (
        k -> (a.getOrElse(k, MV.zero) |+| b.getOrElse(k, MV.zero))
      )
    )
```

Take a monoid for the value type as parameter

Monoid composition

- Some type constructors can form monoids, if their parameters are monoids
- Example: monoid that merges maps, whose values are a monoid:

```
given mapMergeMonoid[K,V](using MV: Monoid[V]): Monoid[Map[K, V]] with
  def zero = Map[K,V]()
  def combine(a: Map[K, V], b: Map[K, V]) =
    (a.keySet ++ b.keySet).foldLeft(zero) ( (acc,k) =>
      acc + (
        k -> (a.getOrElse(k, MV.zero) |+| b.getOrElse(k, MV.zero))
      )
    )
```

Empty map is neutral for merge operation

Monoid composition

- Some type constructors can form monoids, if their parameters are monoids
- Example: monoid that merges maps, whose values are a monoid:

```
given mapMergeMonoid[K,V](using MV: Monoid[V]): Monoid[Map[K, V]] with
  def zero = Map[K,V]()
  def combine(a: Map[K, V], b: Map[K, V]) =
    (a.keySet ++ b.keySet).foldLeft(zero) ( (acc,k) =>
      acc + (
        k -> (a.getOrElse(k, MV.zero) |+| b.getOrElse(k, MV.zero))
      )
    )
```

Fold over the keys of both sets (`++` is normal union of sets)

Monoid composition

- Some type constructors can form monoids, if their parameters are monoids
- Example: monoid that merges maps, whose values are a monoid:

```
given mapMergeMonoid[K,V](using MV: Monoid[V]): Monoid[Map[K, V]] with
  def zero = Map[K,V]()
  def combine(a: Map[K, V], b: Map[K, V]) =
    (a.keySet ++ b.keySet).foldLeft(zero) ( (acc,k) =>
      acc + (
        k -> (a.getOrElse(k, MV.zero) |+| b.getOrElse(k, MV.zero))
      )
    )
```

For each key, get value from both maps. If not present, take zero from value monoid.

Then combine with monoid `combine` and put in accumulator map.

Exercise: Bag of elements

A bag is like a set with a count of occurrences for each element. We represent it by a map from element to count. Example:

```
bag(List("a", "rose", "is", "a", "rose")) ==
  Map("a" -> 2, "rose" -> 2, "is" -> 1)
```

Implement **bag** using monoids:

```
def bag[A](as: IndexedSeq[A]): Map[A, Int]
```

Hint: Use **mapMergeMonoid** and **intAddition**.

Exercise: Bag of elements — Solution

```
def bag[A](as: List[A]): Map[A, Int] =
  combineAll(as.map((a: A) => Map(a -> 1)))(
    using mapMergeMonoid(using intAddition)
  )
```

If we make `intAddition` a given instance (or change it to the typeclass syntax), we don't even have to handle selecting the monoids:

```
given Monoid[Int] = intAddition
def bag[A](as: List[A]): Map[A, Int] =
  combineAll(as.map((a: A) => Map(a -> 1)))
```

Monoids: summary

- We started working with more abstract structures
- Monoids are a simple, but compositional and ubiquitous structure
- We saw useful functions that knew nothing about arguments except monoidical structure
- Up next: more abstractions for common patterns we already saw