|  |  |
| --- | --- |
| **5: Monoids and Semigroups** |  |

Just what is a monoid? It is simply a type together with a binary operation over that type, satisfying associativity and having an identity element. A semigroup is just the combine part of a monoid. While many semigroups are also monoids, there are some data types for which we cannot define an empty element. Like all other abstractions the monoid and the semigroup allow us to write useful generic code assuming only the capabilities provided by these abstractions.

**5.1 Semigroup abstraction**

A *semigroup* for some given type A has a single associative operation (which we will call **combine**), which takes two values of type A, and returns a value of type A. The **Semigroup** is presented with the Kotlin interface:

interface Semigroup<A> {

fun combine(a: A, b: A): A

} // Semigroup

The associativity of operation **combine** means that:

combine(x, combine(y, z)) == combine(combine(x, y), z)

Addition of **Int**s is a binary operation that is closed, meaning that adding two **Int**s produces another **Int**. Similarly, multiplication of **Int**s is a closed binary operation, meaning that multiplying two **Int**s produces another **Int**. Both addition and multiplication are associative operations.

Consequently, integer addition and integer multiplication form semigroups. Example 01a presents an instance of **Semigroup** for integer addition. This instance and many others are supported by the custom Dogs library.

**Example 01a**: *Integer addition semigroup*

package example01a

import com.adt.kotlin.dogs.data.immutable.semigroup.Semigroup

import kotlin.test.\*

val intAddSemigroup: Semigroup<Int> = object: Semigroup<Int> {

override fun combine(a: Int, b: Int): Int = a + b

}

fun main(args: Array<String>) {

assertEquals(7, intAddSemigroup.combine(3, 4))

assertEquals(21, intAddSemigroup.combine(7, 14))

}

The immutable **List** class from the Dogs library is also a semigroup. Example 01b presents two examples.

**Example 01b**: *List semigroup*

package example01b

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.instances.ListSemigroup

import kotlin.test.\*

fun main(args: Array<String>) {

val intListSemigroup = ListSemigroup<Int>()

val stringListSemigroup = ListSemigroup<String>()

assertEquals(

ListF.of(1, 2, 3, 4, 5, 6),

intListSemigroup.combine(ListF.of(1, 2, 3, 4), ListF.of(5, 6))

)

assertEquals(

ListF.of("Ken", "John", "Jessie"),

stringListSemigroup.combine(ListF.of("Ken"), ListF.of("John", "Jessie"))

)

}

The Dogs library provides many semigroup instances out-of-the-box such as the **intAddSemigroup**. It also provides instances for generic types such as **List** regardless of the type parameter. Dogs also provides instances of types that depend on their type parameters. Here is **OptionSemigroup** from the Dogs library. Significantly, we require a semigroup over the type parameter to be able to define the **Option** semigroup.

interface OptionSemigroup<A : Any> : Semigroup<Option<A>> {

val sga: Semigroup<A>

override fun combine(a: Option<A>, b: Option<A>): Option<A> {

return when (a) {

is Option.None -> b

is Option.Some -> {

when (b) {

is Option.None -> a

is Option.Some -> Option.Some(sga.run { combine(a.value, b.value) })

}

}

}

} // combine

} // OptionSemigroup

Example 01c presents using the **OptionSemigroup**. Note how we override the semigroup for the **Int**s within the **OptionSemigroup** wrapper.

**Example 01c**: *Option semigroup*

package example01c

import com.adt.kotlin.dogs.data.immutable.option.OptionF.none

import com.adt.kotlin.dogs.data.immutable.option.OptionF.some

import com.adt.kotlin.dogs.data.immutable.semigroup.Semigroup

import com.adt.kotlin.dogs.data.instances.OptionSemigroup

import com.adt.kotlin.dogs.data.instances.intAddSemigroup

import kotlin.test.\*

fun main(args: Array<String>) {

val intOptionSemigroup = object: OptionSemigroup<Int> {

override val sga: Semigroup<Int> =

intAddSemigroup

}

assertEquals(some(7), intOptionSemigroup.combine(some(3), some(4)))

assertEquals(some(4), intOptionSemigroup.combine(none(), some(4)))

assertEquals(some(3), intOptionSemigroup.combine(some(3), none()))

assertEquals(none(), intOptionSemigroup.combine(none(), none()))

}

Consider a function that merges two **Map**s combining the values if they share the same key. Example 01d declares the function **combineMaps** that merges the two **Map** parameters using the **Semigroup** parameter to perform combining the values. The function utilises the **MapSemigroup** interface. The associated **combine** function folds together the two **Map**s (with **foldLeftWithKey**). If both **Map**s contain the same key then the corresponding values are combined using the **Semigroup** parameter.

**Example 01d**: *Merging Maps*

package example01d

import com.adt.kotlin.dogs.data.immutable.map.Map

import com.adt.kotlin.dogs.data.immutable.map.MapF

import com.adt.kotlin.dogs.data.immutable.map.insert

import com.adt.kotlin.dogs.data.immutable.option.Option

import com.adt.kotlin.dogs.data.immutable.semigroup.Semigroup

import com.adt.kotlin.dogs.data.instances.intAddSemigroup

import kotlin.math.max

import kotlin.test.\*

interface MapSemigroup<K : Comparable<K>, V : Any> : Semigroup<Map<K, V>> {

val sgv: Semigroup<V>

override fun combine(a: Map<K, V>, b: Map<K, V>): Map<K, V> {

return a.foldLeftWithKey(b){map: Map<K, V>, k: K, v: V ->

val ov: Option<V> = map.lookUpKey(k)

ov.fold({map.insert(k, v)}, {vv: V -> map.insert(k, sgv.combine(v, vv))})

}

}

} // MapSemigroup

fun <K : Comparable<K>, V : Any> combineMaps(map1: Map<K, V>, map2: Map<K, V>, sgv: Semigroup<V>): Map<K, V> {

val mapSemigroup = object: MapSemigroup<K, V> {

override val sgv: Semigroup<V> = sgv

}

return mapSemigroup.combine(map1, map2)

} // combineMaps

fun main(args: Array<String>) {

val intMaxSemigroup = object: Semigroup<Int> {

override fun combine(a: Int, b: Int): Int = max(a, b)

}

val map1 = MapF.of("a" to 1, "b" to 2, "c" to 8)

val map2 = MapF.of("b" to 3, "c" to 4, "d" to 2)

val addMaps = combineMaps(map1, map2, intAddSemigroup)

val maxMaps = combineMaps(map1, map2, intMaxSemigroup)

assertEquals(MapF.of("a" to 1, "b" to 5, "c" to 12, "d" to 2), addMaps)

assertEquals(MapF.of("a" to 1, "b" to 3, "c" to 8, "d" to 2), maxMaps)

}

Here, the **Semigroup** abstraction lets us write useful generic code assuming only the capabilities provided by the abstraction. In function **main** we combine **Map**s adding the values of matching keys, and combine **Map**s associating the key with the largest value.

The Dogs library defines instances of the **Semigroup** interface for many of types supported by the library. We know that the **combine** operation of a **Semigroup** must be associative. Using KwikCheck we can test the associativity rule for the instances. Example 01e tests the associativity law for the integer addition, string and list **Semigroup**s. The final test function confirms that integer subtraction does not form a **Semigroup** since

(x – y) – z != x – (y – z)

A concrete illustration would be:

left side: (10 – 6) – 1 = 3

right side: 10 – (6 – 1) = 5

**Example 01e**: *Associativity law*

package example01e

import com.adt.kotlin.dogs.data.immutable.semigroup.Semigroup

import com.adt.kotlin.dogs.data.instances.ListSemigroup

import com.adt.kotlin.dogs.data.instances.intAddSemigroup

import com.adt.kotlin.dogs.data.instances.stringSemigroup

import com.adt.kotlin.kwikcheck.generator.GenF.genInt

import com.adt.kotlin.kwikcheck.generator.GenF.genList

import com.adt.kotlin.kwikcheck.generator.GenF.genPosInt

import com.adt.kotlin.kwikcheck.generator.GenF.genString

import com.adt.kotlin.kwikcheck.property.Property

import com.adt.kotlin.kwikcheck.property.PropertyF.forAll

import com.adt.kotlin.kwikcheck.property.PropertyF.prop

import org.junit.Test

import kotlin.test.\*

object SemigroupLaws {

fun <A> associativityLaw(sg: Semigroup<A>, a1: A, a2: A, a3: A): Boolean {

return sg.combine(sg.combine(a1, a2), a3) == sg.combine(a1, sg.combine(a2, a3))

} // associativityLaw

} // SemigroupLaws

class Example01e {

@Test

fun intAdditionSemigroupAssociativityOperation() {

val property: Property = forAll(genPosInt, genPosInt, genPosInt) { n1, n2, n3 ->

prop(SemigroupLaws.associativityLaw(intAddSemigroup, n1, n2, n3))

}

val checkResult = property.check()

assertTrue(checkResult.isPassed())

} // intAdditionSemigroupAssociativityOperation

@Test

fun stringSemigroupAssociativityOperation() {

val property: Property = forAll(genString, genString, genString) {str1, str2, str3 ->

prop(SemigroupLaws.associativityLaw(stringSemigroup, str1, str2, str3))

}

val checkResult = property.check()

assertTrue(checkResult.isPassed())

} // stringSemigroupAssociativityOperation

@Test

fun listSemigroupAssociativityOperation() {

val property: Property = forAll(genList(genInt), genList(genInt), genList(genInt)) { ls1, ls2, ls3 ->

prop(SemigroupLaws.associativityLaw(ListSemigroup(), ls1, ls2, ls3))

}

val checkResult = property.check()

assertTrue(checkResult.isPassed())

} // listSemigroupAssociativityOperation

@Test

fun intSubtractionSemigroupAssociativityOperation() {

val intSubSemigroup: Semigroup<Int> = object: Semigroup<Int> {

override fun combine(a: Int, b: Int): Int = a - b

}

val property: Property = forAll(genPosInt, genPosInt, genPosInt) { n1, n2, n3 ->

prop(SemigroupLaws.associativityLaw(intSubSemigroup, n1, n2, n3))

}

val checkResult = property.check()

assertFalse(checkResult.isPassed())

} // intSubtractionSemigroupAssociativityOperation

}

The above examples have demonstrated the utility of the **Semigroup**. However, we quickly run into issues if we try to write a generic **combineAll** function:

fun <A> List<A>.combineAll(sg: Semigroup<A>): A =

this.foldLeft(/\* what goes here?? \*/) { a: A, b: A -> sg.combine(a, b) }

Here, the **Semigroup** does not provide us with the required initial value. We need the more expressive **Monoid** abstraction.

**5.2 Monoid abstraction**

The **Monoid** abstraction extends the **Semigroup** abstraction by providing an additional **empty** value:

interface Semigroup<A> {

fun combine(a: A, b: A): A

} // Semigroup

interface Monoid<A : Any> : Semigroup<A> {

val empty: A

} // Monoid

The **empty** value should be an *identity element* for the **combine** operation, which means the following equalities should hold for any **a** value:

combine(a, empty) = a

combine(empty, a) = a

Many types that form a **Semigroup** also form a **Monoid**, such as **Int**s (with 0 or 1 as the identity element) or *String*s (with the empty string “” as the identity element). Example 02a demonstrates using the **intAddMonoid** and the **stringMonoid** from the Dogs library.

**Example 02a**: *Simple monoids*

package example02a

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.monoid.Monoid

import com.adt.kotlin.dogs.data.instances.intAddMonoid

import com.adt.kotlin.dogs.data.instances.stringMonoid

import kotlin.test.\*

fun <A : Any> List<A>.combineAll(ma: Monoid<A>): A =

this.foldLeft(ma.empty, ma::combine)

fun main(args: Array<String>) {

assertEquals(10, ListF.of(1, 2, 3, 4).combineAll(intAddMonoid))

assertEquals("hello world", ListF.of("hello", " ", "world").combineAll(stringMonoid))

}

The **Option** type also forms a monoid. Example 02b presents two examples: an **Option** monoid over **Int**s and an **Option** monoid over **String**s.

**Example 02b**: *Option monoid*

package example02b

import com.adt.kotlin.dogs.data.immutable.option.OptionF.none

import com.adt.kotlin.dogs.data.immutable.option.OptionF.some

import com.adt.kotlin.dogs.data.instances.\*

import kotlin.test.assertEquals

val intAddOptionMonoid = OptionMonoid(intAddSemigroup)

val stringOptionMonoid = OptionMonoid(stringSemigroup)

fun main(args: Array<String>) {

assertEquals(some(5), intAddOptionMonoid.combine(none(), some(5)))

assertEquals(some(9), intAddOptionMonoid.combine(some(4), some(5)))

assertEquals(some("Barclay"), stringOptionMonoid.combine(none(), some("Barclay")))

assertEquals(some("KenBarclay"), stringOptionMonoid.combine(some("Ken"), some("Barclay")))

}

Note how the **OptionMonoid** constructor requires an appropriate **Semigroup** instance. To have an Option<A> as a monoid requires that its type parameter A is a **Semigroup**. In the implementation for **OptionMonoid** we see how the **Semigroup** parameter is used to combine the two values wrapped in **Some** instances:

class OptionMonoid<A : Any>(val sga: Semigroup<A>) : Monoid<Option<A>> {

override val empty: Option<A> = Option.None

override fun combine(a: Option<A>, b: Option<A>): Option<A> {

return when (a) {

is Option.None -> b

is Option.Some -> {

when (b) {

is Option.None -> a

is Option.Some -> Option.Some(sga.combine(a.value, b.value))

}

}

}

} // combine

} // OptionMonoid

The **List** type includes the function **foldMap** which maps each element of the **List** to a monoid and combines the result. Its signature is:

fun <A : Any, B : Any> List<A>.foldMap(md: Monoid<B>, f: (A) -> B): B

To use this function with a **Pair**, we define a monoid over a **Pair** where the two element types are also monoids. We see this in the class **PairMonoid**:

class PairMonoid<A : Any, B : Any>(val ma: Monoid<A>, val mb: Monoid<B>) : Monoid<Pair<A, B>> {

override val empty: Pair<A, B> = Pair(ma.run{ empty }, mb.run{ empty })

override fun combine(a: Pair<A, B>, b: Pair<A, B>): Pair<A, B> {

val first: A = ma.run { combine(a.first, b.first) }

val second: B = mb.run{ combine(a.second, b.second) }

return Pair(first, second)

} // combine

} // PairMonoid

Example 02c contains an example of using **foldMap** with a **Pair**.

**Example 02c**: *List foldMap*

package example02c

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.list.foldMap

import com.adt.kotlin.dogs.data.instances.\*

import kotlin.test.assertEquals

fun main(args: Array<String>) {

val list: List<Int> = ListF.of(1, 2, 3, 4, 5)

assertEquals(

Pair(15, "12345"),

list.foldMap(PairMonoid(intAddMonoid, stringMonoid)){ n -> Pair(n, n.toString())}

)

}

At the end of the previous section we were unable to define a generic **combineAll** function over a **List** because we had no value to use when the **List** is empty. With a monoid we can develop **combineAll** as shown in Example 02d.

There are some types that form a **Semigroup** but not a **Monoid**. For example the **NonEmptyList** from the Dogs library forms a **Semigroup** through its **concatenate** member function, but has no corresponding identity element to form a monoid. How might we collapse a List<NonEmptyList<A>> using **combineAll** when the **NonEmptyList** only has a **Semigroup**? We have already seen OptionMonoid<A> defined where we have Semigroup<A>. Thus if we wrap the **NonEmptyList**s contained in the **List**, producing a List<Option<NonEmptyList<A>>> we can run **combineAll** with a suitable **Option** monoid>

**Example 02d**: *Monoid wrapping a NonEmptyList*

package example02d

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.monoid.Monoid

import com.adt.kotlin.dogs.data.immutable.nel.NonEmptyList

import com.adt.kotlin.dogs.data.immutable.nel.NonEmptyListF

import com.adt.kotlin.dogs.data.immutable.option.Option

import com.adt.kotlin.dogs.data.immutable.option.OptionF.some

import com.adt.kotlin.dogs.data.instances.NonEmptyListSemigroup

import com.adt.kotlin.dogs.data.instances.OptionMonoid

import kotlin.test.assertEquals

fun <A : Any> List<A>.combineAll(ma: Monoid<A>): A =

this.foldLeft(ma.empty, ma::combine)

fun <A : Any> optionNelMonoid(): Monoid<Option<NonEmptyList<A>>> =

OptionMonoid(NonEmptyListSemigroup())

fun main(args: Array<String>) {

val listNel: List<NonEmptyList<Int>> = ListF.of(NonEmptyListF.of(1, 2, 3), NonEmptyListF.of(4, 5))

val opListNel: List<Option<NonEmptyList<Int>>> = listNel.map { nel -> some(nel) }

assertEquals(some(NonEmptyListF.of(1, 2, 3, 4, 5)), opListNel.combineAll(optionNelMonoid()))

}

The **combineAll** function can be used with any type that has a monoid instance. In Example 02e we define a **Monoid** over the **Map** type from the Dogs library. The redefined function **combine** performs a left fold (strictly **foldLeftWithKey**) on the first **Map** with the second **Map** as initial value. If the given key is not a member of the given accumulating **Map** then the key/value pair is inserted into the accumulator; otherwise the key and combined values are inserted.

**Example 02e**: *Combining Maps*

package example02e

import com.adt.kotlin.dogs.data.immutable.list.List

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.map.Map

import com.adt.kotlin.dogs.data.immutable.map.MapF

import com.adt.kotlin.dogs.data.immutable.map.insert

import com.adt.kotlin.dogs.data.immutable.monoid.Monoid

import com.adt.kotlin.dogs.data.immutable.option.Option

import com.adt.kotlin.dogs.data.immutable.semigroup.Semigroup

import com.adt.kotlin.dogs.data.instances.intAddSemigroup

import kotlin.test.\*

interface MapMonoid<K : Comparable<K>, V : Any> : Monoid<Map<K, V>> {

val sgv: Semigroup<V>

override fun combine(a: Map<K, V>, b: Map<K, V>): Map<K, V> {

return a.foldLeftWithKey(b){map: Map<K, V>, k: K, v: V ->

val ov: Option<V> = map.lookUpKey(k)

ov.fold({map.insert(k, v)}, {vv: V -> map.insert(k, sgv.combine(v, vv))})

}

}

} // MapMonoid

fun <A : Any> List<A>.combineAll(ma: Monoid<A>): A =

this.foldLeft(ma.empty, ma::combine)

fun main(args: Array<String>) {

val mapMonoid = object: MapMonoid<Char, Int> {

override val empty: Map<Char, Int> = MapF.empty()

override val sgv: Semigroup<Int> = intAddSemigroup

}

assertEquals(

MapF.of('a' to 3, 'b' to 7, 'c' to 5),

ListF.of(MapF.of('a' to 1), MapF.of('a' to 2, 'b' to 3), MapF.of('b' to 4, 'c' to 5)).combineAll(mapMonoid)

)

}

Consider a user defined class representing a monetary amount. Given a **List** of these amounts we can fold left across the **List** provided we have a monoid for the monetary amount. This is shown in Example 02f.

**Example 02f**: *Money monoid*

package example02f

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.monoid.Monoid

import kotlin.test.assertEquals

data class Money(val dollars: Int, val cents: Int)

val moneyMonoid: Monoid<Money> = object: Monoid<Money> {

override val empty: Money = Money(0, 0)

override fun combine(a: Money, b: Money): Money =

Money(a.dollars + b.dollars + (a.cents + b.cents) / 100, (a.cents + b.cents) % 100)

}

fun main(args: Array<String>) {

val expenses = ListF.of(Money(3, 4), Money(5, 6), Money(7, 8))

val totalExpenses = expenses.foldLeft(moneyMonoid.empty){acc, money -> moneyMonoid.combine(acc, money)}

assertEquals(Money(15, 18), totalExpenses)

}

Example 02g develops another monoid for a user defined type. The **User** class represents a user of a social media application. It is comparable with other **User** instances through the **compareTo** function. The primary comparison is based on the number of followers otherwise defaulting to their names. The **User** monoid defines **combine** to select the greater of the two **User**s. The example finds the larger of a **List** of **User**s.

**Example 02g**: *User monoid*

package example02g

import com.adt.kotlin.dogs.data.immutable.list.ListF

import com.adt.kotlin.dogs.data.immutable.monoid.Monoid

import kotlin.test.assertEquals

data class User(val name: String, val followers: Int) : Comparable<User> {

override fun compareTo(other: User): Int {

val n = this.followers - other.followers

return if (n == 0) this.name.compareTo(other.name) else n

}

} // User

val userMonoid = object: Monoid<User> {

override val empty: User = User("Minimum", 0)

override fun combine(a: User, b: User): User =

if (a.compareTo(b) >= 1) a else b

}

fun main(args: Array<String>) {

val users = ListF.of(User("Ken", 100), User("John", 200), User("Jessie", 150))

assertEquals(User("John", 200), users.foldLeft(userMonoid.empty){acc, user -> userMonoid.combine(acc, user)})

}

Example 02h presents the associativity law, and the left and right identity laws for the **Monoid**. The associativity law is inherited through the **Semigroup**. The left and right identity laws were presented earlier.

**Example 02h**: *Monoid laws*

package example02h

import com.adt.kotlin.dogs.data.immutable.monoid.Monoid

import com.adt.kotlin.dogs.data.instances.ListMonoid

import com.adt.kotlin.dogs.data.instances.intAddMonoid

import com.adt.kotlin.dogs.data.instances.stringMonoid

import com.adt.kotlin.kwikcheck.generator.GenF.genList

import com.adt.kotlin.kwikcheck.generator.GenF.genPosInt

import com.adt.kotlin.kwikcheck.generator.GenF.genString

import com.adt.kotlin.kwikcheck.property.Property

import com.adt.kotlin.kwikcheck.property.PropertyF.forAll

import com.adt.kotlin.kwikcheck.property.PropertyF.prop

import org.junit.Test

import kotlin.test.\*

object MonoidLaws {

fun <A : Any> associativityLaw(md: Monoid<A>, a1: A, a2: A, a3: A): Boolean {

return md.combine(md.combine(a1, a2), a3) == md.combine(a1, md.combine(a2, a3))

} // associativityLaw

fun <A : Any> rightIdentityLaw(md: Monoid<A>, a: A): Boolean {

return md.combine(a, md.empty) == a

} // rightIdentityLaw

fun <A : Any> leftIdentityLaw(md: Monoid<A>, a: A): Boolean {

return md.combine(md.empty, a) == a

} // leftIdentityLaw

} // MonoidLaws

class Example02h {

@Test fun intMonoidOperation() {

val associativityProperty: Property = forAll(genPosInt, genPosInt, genPosInt) { n1, n2, n3 ->

prop(MonoidLaws.associativityLaw(intAddMonoid, n1, n2, n3))

}

val rightIdentityProperty: Property = forAll(genPosInt) { n ->

prop(MonoidLaws.rightIdentityLaw(intAddMonoid, n))

}

val leftIdentityProperty: Property = forAll(genPosInt) { n ->

prop(MonoidLaws.leftIdentityLaw(intAddMonoid, n))

}

val property: Property = associativityProperty and rightIdentityProperty and leftIdentityProperty

val checkResult = property.check()

assertTrue(checkResult.isPassed())

}

@Test fun stringMonoidOperation() {

val associativityProperty: Property = forAll(genString, genString, genString) { str1, str2, str3 ->

prop(MonoidLaws.associativityLaw(stringMonoid, str1, str2, str3))

}

val rightIdentityProperty: Property = forAll(genString) { str ->

prop(MonoidLaws.rightIdentityLaw(stringMonoid, str))

}

val leftIdentityProperty: Property = forAll(genString) { str ->

prop(MonoidLaws.leftIdentityLaw(stringMonoid, str))

}

val property: Property = associativityProperty and rightIdentityProperty and leftIdentityProperty

val checkResult = property.check()

assertTrue(checkResult.isPassed())

}

@Test fun listMonoidOperation() {

val associativityProperty: Property = forAll(genList(genString), genList(genString), genList(genString)) { list1, list2, list3 ->

prop(MonoidLaws.associativityLaw(ListMonoid(), list1, list2, list3))

}

val rightIdentityProperty: Property = forAll(genList(genString)) { list ->

prop(MonoidLaws.rightIdentityLaw(ListMonoid(), list))

}

val leftIdentityProperty: Property = forAll(genList(genString)) { list ->

prop(MonoidLaws.leftIdentityLaw(ListMonoid(), list))

}

val property: Property = associativityProperty and rightIdentityProperty and leftIdentityProperty

val checkResult = property.check()

assertTrue(checkResult.isPassed())

}

}