|  |  |
| --- | --- |
| **7: Monads** |  |

Engineering in general, and software engineering in particular, is all about *abstraction*. The creation and utilization of abstractions is a core part of the daily activities of every good software engineer, and many engineers are on a never ending quest to increase their level of abstraction. Abstraction both increases productivity and increases the complexity of problems that can be tackled. Few would argue that increased abstraction is a bad thing.

The *monad* can prove a rather slippery abstraction. A monad is a way to structure computations in terms of values and sequences of computations using those values. Monads allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required. It is useful to think of a monad as a strategy for combining computations into more complex computations.

A monadic library will have some representation of primitive computations, and some ways to glue those computations together into more complex ones. Our monadic library defines primitive combinators, and then combining functions for concatenating these combinators or for selecting between them. In this manner, the user of the combinator library builds up the computation they want, piecing together smaller parts into larger ones.

This chapter presents a range of monadic libraries and numerous examples to illustrate their capabilities. As a simple example we present the option type as a monad. An advanced monadic library is shown with the data generator at the core of the KwikCheck framework.

**7.1 Monad abstraction**

The *monad* is the constructor type **F** that supports the functions **bind** (aka **flatMap**) and **inject**:

fun <A, B> **F**<A>.bind(f: (A) -> **F**<B>): **F**<B>

fun <A> inject(a: A): **F**<A>

The basic intuition behind the **bind** operation is that it allows us to combine two computations into one more complex computation and allows them to interact with one another. The first parameter type **F**<A> represents the first computation. Significantly, the second parameter type is A -> **F**<B> which can, given the result of the first computation, produce a second computation to run. In other words bind(ma){a -> ... } is a computation which runs **ma** and binds the value wrapped in **ma** to the function parameter **a**. The function body then decides what computation to run using the value for **a**.

Function **inject** lets us put a value into a monadic context. For example, for the list type **inject** is the same as creating a list with a single element. Often **inject** is simply the class constructor.

Example 01a presents samples of the monadic **bind** applied to **Option**, **Either** and **List** types. The first two asserts apply the **bind** to an Option<Int>. The function n -> some(1 + n) takes an **Int**, adds 1 to it and wraps the result in a **Some**. When we feed it some(7) it returns some(8). When we feed it none() it returns none(). When we try using it with none() the result is none(). The third and fourth asserts shows what happens if the **bind** function were to return none().

**Example 01a**: *Monadic bind*

package example01a

import com.adt.kotlin.dogs.data.immutable.either.EitherF.left

import com.adt.kotlin.dogs.data.immutable.either.EitherF.right

import com.adt.kotlin.dogs.data.immutable.either.bind

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

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

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.option.bind

import kotlin.test.\*

fun main(args: Array<String>) {

assertEquals(some(8), some(7).bind { n -> some(1 + n) })

assertEquals(none(), none<Int>().bind { n -> some(1 + n) })

assertEquals(some(3), some(3).bind { n -> if (n > 2) some(n) else none() })

assertEquals(none(), some(1).bind { n -> if (n > 2) some(n) else none() })

assertEquals(right(16), right<String, Int>(4).bind { n -> right(n \* n) })

assertEquals(left("BUG"), left<String, Int>("BUG").bind { n -> right(n \* n) })

assertEquals(

ListF.of(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5),

ListF.of(1, 2, 3, 4).bind { n -> ListF.of(n - 1, n, n + 1) }

)

}

The next two asserts apply the monadic **bind** to Either<String, Int> instances. We see that the function operates on a **Right** instance and returns unchanged a **Left** instance. Because of this **Either** is said to be *right-biased*.

The final assert shows the **bind** applied to a **List**. For each element n in the source **List** a new **List** containing n – 1, n and n + 1. Hence the first element ListF.of(1, …) produces the ListF.of(0, 1, 2). This might suggest that the final outcome is ListF.of(ListF.of(0, 1, 2), ListF.of(1, 2, 3), ListF.of(2, 3, 4), ListF.of(3, 4, 5)). However, the nested lists are flattened (hence the synonym **flatMap** for **bind**) producing the shown result.

Example 01b demonstrates how we use the monadic **Option** type. Running the program with the inputs 18, 27 and 9 delivers the result Some(Pair(2, 3)), demonstrating that 18 and 27 are exactly divisible by 9. The inputs 18, 27 and 6 produce the result **None**, showing that 18 and 27 and not both exactly divisible by 6.

**Example 01b**: *Staircasing*

package example01b

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

import com.adt.kotlin.dogs.data.immutable.option.Option.None

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

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.option.bind

import kotlin.test.\*

fun divide(num: Int, den: Int): Option<Int> {

return if (num % den != 0) none() else some(num / den)

}

fun division(a: Int, b: Int, c: Int): Option<Pair<Int, Int>> {

return when (val ac = divide(a, c)) {

is None -> none()

is Some -> when (val bc = divide(b, c)) {

is None -> none()

is Some -> some(Pair(ac.get(), bc.get()))

}

}

} // division

fun bindDivision(a: Int, b: Int, c: Int): Option<Pair<Int, Int>> =

divide(a, c).bind { ac ->

divide(b, c).bind { bc ->

some(Pair(ac, bc))

}

}

fun main(args: Array<String>) {

assertEquals(some(Pair(2, 3)), division(18, 27, 9))

assertEquals(none(), division(18, 27, 6))

assertEquals(some(Pair(2, 3)), bindDivision(18, 27, 9))

assertEquals(none(), bindDivision(18, 27, 6))

}

Function **division** threatens to march off to the right side of the listing if it became any more complicated. We can bring the staircasing effect under control with **bind**. Consider the implementation of **bindDivision**. The outer call to **divide(a, c).bind{ ac -> ...}** has an **Option** value produced from the expression **divide(a, c)**. Should this be a **Some** value, then the function literal is called and its formal parameter **ac** is bound to the result from the **Some** value. In the inner call **divide(b, c).bind{ bc -> ...}** the inner function literal is called with the formal parameter **bc** bound to the **Some** value produced from **divide(b, c)**. If the two calls to **divide** both deliver **Some** values then a final **Some** result is produced carrying the pair we seek. If a **None** value is delivered by either call to **divide** then a **None** value is the final result.

**7.2 List monad**

Just as **List** is a functor, it is also a monad. Consequently, we expect declarations for the following functions:

fun <A> List<A>.bind(f: (A) -> List<B>): List<B>

fun <A> inject(a: A): List<A>

The **inject** function for **List**s simply inserts a value into a **List** using the **List** singleton operation. The **bind** function pulls out the values from the source **List** to give them to a function that creates a new **List**. Of course, as we described above, the result is flattened.

The first assert in Example 02a takes the elements of a **List** of **Int**s and produces a **List** with the element value and it’s negative. In the second assert the two strings in the source **List** are split around the comma and added into a new **List**. In the third assert we apply the monadic bind to a **List** of **String**s and the monadic **bind** to a **List** of **Int**s. If any of the **Int** values are even-valued, then it is paired with the bound **String**.

**Example 02a**: *List monad*

package example02a

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

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

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

import com.adt.kotlin.dogs.fp.FunctionF.isEven

import kotlin.test.\*

fun main(args: Array<String>) {

assertEquals(ListF.of(-1, 1, -2, 2, -3, 3, -4, 4), ListF.of(1, 2, 3, 4).bind { n -> ListF.of(-n, +n) })

assertEquals(

ListF.of("one", "two", "three", "four", "five", "six", "seven"),

ListF.of("one,two,three", "four,five,six,seven").bind { str -> ListF.from(str.split(",")) }

)

assertEquals(

ListF.of(Pair("one", 2), Pair("one", 4), Pair("two", 2), Pair("two", 4), Pair("three", 2), Pair("three", 4)),

ListF.of("one", "two", "three").bind { str -> ListF.of(1, 2, 3, 4).bind { n -> if (isEven(n)) inject(Pair(str, n)) else ListF.empty() } }

)

assertEquals(ListF.of(1, 2, 3, 4), ListF.of(1, 2, 3, 4).bind { n -> inject(n) })

assertEquals(

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

ListF.of(1, 2, 3).bind { m -> ListF.of(4, 5).bind { n -> inject(Pair(m, n)) } }

)

}

The ubiquity of this **bind** pattern becomes more apparent when we start thinking about iterating over multiple **List**s. For example, suppose now that we have two lists and that we want to iterate over all pairs of elements consisting of one element from each list. Example 02b presents two examples. In, for example, the second assert the list [1, 2, 3] is bound to **n** and the list [a, b] is bound to **ch** and finish by returning the pair (n, ch).

**Example 02b**: *Iterating over multiple lists*

package example02b

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

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

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

import kotlin.test.\*

fun main(args: Array<String>) {

val list1 = ListF.of(1, 2, 3)

val list2 = ListF.of(4, 5, 6, 7)

assertEquals(

ListF.of(4, 5, 6, 7, 8, 10, 12, 14, 12, 15, 18, 21),

list1.bind { n1 -> list2.bind { n2 -> inject(n1 \* n2) } }

)

assertEquals(

ListF.of(Pair(1, 'a'), Pair(1, 'b'), Pair(2, 'a'), Pair(2, 'b'), Pair(3, 'a'), Pair(3, 'b')),

ListF.of(1, 2, 3).bind { n -> ListF.of('a', 'b').bind { ch -> inject(Pair(n, ch)) } }

)

}

The pattern of having one or more nested **bind**s followed by a final **fmap** in order to iterate over multiple collections is very common. It is exactly this pattern that the **for2**, **for3**, etc. library functions are for. So we can re-write the above using a **for2** function call. It has the signature:

fun <A, B, C> for2(la: List<A>, lb: List<B>, f: (A) -> (B) -> C): List<C>

**Example 02c**: *for2 function*

package example02c

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

import com.adt.kotlin.dogs.data.immutable.list.ListF.for2

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

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

import kotlin.test.\*

fun main(args: Array<String>) {

val list1 = ListF.of(1, 2, 3)

val list2 = ListF.of(4, 5, 6, 7)

assertEquals(

ListF.of(4, 5, 6, 7, 8, 10, 12, 14, 12, 15, 18, 21),

list1.bind { n1 -> list2.fmap { n2 -> n1 \* n2 } }

)

assertEquals(

ListF.of(Pair(1, 'a'), Pair(1, 'b'), Pair(2, 'a'), Pair(2, 'b'), Pair(3, 'a'), Pair(3, 'b')),

ListF.of(1, 2, 3).bind { n -> ListF.of('a', 'b').fmap { ch -> Pair(n, ch) } }

)

assertEquals(

ListF.of(4, 5, 6, 7, 8, 10, 12, 14, 12, 15, 18, 21),

for2(list1, list2) { n1, n2 -> n1 \* n2 }

)

assertEquals(

ListF.of(Pair(1, 'a'), Pair(1, 'b'), Pair(2, 'a'), Pair(2, 'b'), Pair(3, 'a'), Pair(3, 'b')),

for2(ListF.of(1, 2, 3), ListF.of('a', 'b')) { n, ch -> Pair(n, ch) }

)

}

**7.3 Monad laws**

The **List** as a monad requires that we should ensure it satisfies the three monad laws: the two *identity laws* and the *associativity law*. If we follow the patterns from the previous chapters we can readily derive the **ListMonadLaws** which we can show are satisfied.

**Example 03**: *List monad laws*

package example03

/\*\*

\* Monads have three laws. The first two are simple identity laws, like our

\* other classes have had:

\*

\* inject(a).bind(f) ==> f

\* x.bind(inject) ==> x

\*

\* These are the left and right identities. They state effectively that the

\* only thing the return function is allowed to do is to wrap the object.

\* It cannot manipulate the data in any way.

\*

\* The third law tells us that associativity holds within monads:

\*

\* x.bind(f).bind(g) ==> x.bind{y -> f(y).bind(g)}

\*

\* We see this third law has a parallel structure to the other composition laws.

\* In the first case, we apply two functions in two steps. In the second case,

\* we compose the functions first, and then apply the result. These should be the same.

\*/

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

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

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

import com.adt.kotlin.kwikcheck.generator.CogenF

import com.adt.kotlin.kwikcheck.generator.Gen

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

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

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

import org.junit.Test

import kotlin.test.\*

object ListMonadLaws {

fun <A : Any, B : Any> leftIdentityLaw(a: A, f: (A) -> List<B>): Boolean {

return (inject(a).bind(f) == f(a))

} // leftIdentityLaw

fun <A : Any> rightIdentityLaw(lsa: List<A>): Boolean {

return (lsa.bind{ a: A -> inject(a) } == lsa)

} // rightIdentityLaw

fun <A : Any, B : Any, C : Any> associativityLaw(lsa: List<A>, f: (A) -> List<B>, g: (B) -> List<C>): Boolean {

return (lsa.bind{ a: A -> f(a).bind{ b: B -> g(b) } } == lsa.bind{ a: A -> f(a) }.bind{ b: B -> g(b)} )

} // associativityLaw

} // ListMonadLaws

class Example03 {

@Test fun listMonadOperation() {

val listLaws = ListMonadLaws

val stringToListInt: Gen<(String) -> List<Int>> = GenF.genF(CogenF.cogenString, GenF.genList(GenF.genInt))

val intToListBoolean: Gen<(Int) -> List<Boolean>> = GenF.genF(CogenF.cogenInt, GenF.genList(GenF.genBoolean))

val property = forAll(

GenF.genString,

stringToListInt,

GenF.genList(GenF.genString),

intToListBoolean

) { str, strToListInt, list, intToListBool ->

val leftIdentity: Boolean = listLaws.leftIdentityLaw(str, strToListInt)

val rightIdentity: Boolean = listLaws.rightIdentityLaw(list)

val associativity: Boolean = listLaws.associativityLaw(list, strToListInt, intToListBool)

prop(leftIdentity && rightIdentity && associativity)

}

val checkResult = property.check()

assertTrue(checkResult.isPassed())

}

}