Declarative Looping in Scala: the For-Comprehension Structure : Functional Sets Building Syntax.

The for-comprehension is the Scala way to manage collections using a purely declarative style:

val listOfPassedAssertsInSucceededTests: List[Int] =

for {

result <- results

if result.succeeded

} yield (result.successfulAsserts)

val passedAssertsInSucceededTests: Int = listOfPassedAssertsInSucceededTests.sum

We can identify different constructs that, when taken together, form the for-comprehension syntax. This might look daunting, but let’s walk through this code and take each component step by step.

The for-comprehension in Scala has the form for (enumerators) yield e. The enumerators are the collective code that goes inside the parentheses that follow the for keyword. Enumerators bind values to variables. The for-comprehension body e evaluates for every value generated by enumerators, creating a sequence of such values.

We’ll take a closer look at each component of the for-comprehension in the next sections.

Enumerators

An enumerator can be either a generator or a filter. In our previous example, we have a for-comprehension containing both types of enumerators. Let’s take a closer look at each type.

4.1. Generators

The statement result <- results represent a generator. It introduces a new variable, result, that loops over each value of the variable results. So, the type of result is TestResult.

We can have as many generators as we want. They loop independently from each other, producing all the possible combinations of their variables. In the example, we loop over the list of results and the list of execution times. Then, we merge the two elements, listing the total number of asserts executed for each test result, along with the execution time:

val executionTimeList = List(("test 1", 100), ("test 2", 230))

val numberOfAssertsWithExecutionTime: List[(String, Int, Int)] =

for {

result <- results

(id, time) <- executionTimeList

if result.id == id

} yield ((id, result.totalAsserts, time))

The values contained in the numberOfAssertsWithExecutionTime list are:

List[("test 1", 10, 100), ("test 2", 6, 230)]

All the generators inside a for-comprehension must share the same type they loop over. In our previous example, both were instances of List. The type variable does not count. So, we can mix a generator of type List[TestResult] with a generator of type List[(String, Int)].

4.2. Filters

Inside a for-comprehension, filters have the form if boolean-condition. A filter acts as a guard that blocks all the values that do not respect the boolean condition.

Inside a filter, we can create a custom boolean condition using every variable that is available in the scope of the for-comprehension.

In the previous examples, we used the variables declared by the generators. However, we can also use variables declared outside the for-comprehension. Let’s see an example:

val hugeNumberOfAssertsForATest: Int = 10

val resultsWithAHugeAmountOfAsserts: List[TestResult] =

for {

result <- results

if result.totalAsserts >= hugeNumberOfAssertsForATest

} yield (result)

5. The For-Comprehension Body

As we said, the for-comprehension body evaluates for every value generated by enumerators, creating a sequence of such values. Inside the body, we can use any variable or value that is in the scope of the for-comprehension:

val magic: Int = 42

for {

res <- result

} yield res \* magic

The type of yield body can be anything we want. Until now, our examples return something as a result of the for-comprehension. However, it’s feasible to return nothing, using an expression that evaluates to Unit. For example, we can use the yield body to print the variables bound by the generators:

for {

res <- result

} println(s"The result is $res")

In the case where the yield body evaluates to Unit, it is possible to omit the yield keyword.

6. For-Comprehension: Deep Dive

In the previous example, we saw how the semantics of a for-comprehension are equal to that of a sequence of operations on streams or sequences. In Scala, the for-comprehension is nothing more than syntactic sugar to a sequence of calls to one or more of the methods:

foreach

map

flatMap

withFilter

We can use for-comprehension syntax on every type that defines such methods.

List Comprehensions in Java:

Build sets in mathematical set-builder notation with Java, like

{ x \* 2 | x E {1,2,3,4} ^ x is even }.

Now in use in the jComprehension java library.

ABSTRACT

In many programming languages, such as Haskell or python, it is popular to write list comprehensions, like we write in algebra

as the list of all even numbers in the set of real numbers. This article proposes an implementation of list comprehensions in Java, and then provides the implementation of the functions map and filter using the proposed code. Also, it supports binding more than one variable to the list’s definition, so we will be able to implement lists of cartesian products and also to add a relation that must hold between the two variables, like

INTRODUCTION

We may remember from algebra that we can define lists as lists comprehensions, which uses a special notation called set-builder notation. For example,

denotes the set of all pairs (x,y) such that x and y are real numbers and the product of both numbers equals the sum.

Today, this syntactic construct is integrated in many programming languages, such as

Haskell

[x \* 2 | x <- [0 .. 99], x \* x > 3]

python

S = [2\*x for x in range(100) if x\*\*2 > 3]

C#

var ns = Enumerable.Range(0, 100) .Where(x => x\*x > 3) .Select(x => x\*2);

Scala

val s = for (x <- Stream.from(0); if x\*x > 3) yield 2\*x

Ruby

(0..100).select { |x| x\*\*2 > 3 } .map { |x| 2\*x }

and more. But the Java language does not provide a syntactic construct for this concept.

The usage of list comprehensions in Haskell motivated this article. Bringing the set-builder notation to Java, this different way of thinking problems. Instead of using for or while statements in your programs, you may work with map and filter functions, but take into account that thinking problems as list comprehensions is also another good way to practice programming.

The structure of this article follows:

1. Set-builder notation and list comprehensions definitions as we saw in algebra.

2. Some words about the implementation of Haskell’s list comprehensions. The implementation of map and filter with list comprehensions in Haskell.

3. The proposed implementation of list comprehensions in Java. The implementation of map, filter and the cartesian product with our new list comprehensions.

4. Discussion. Is this useful at industrial level? Possible enhancements and future work.

Get started

Open in app

Newton’s notebook

List Comprehensions in Java

Franco Arolfo

Franco Arolfo

Dec 6, 2015·7 min read

Build sets in mathematical set-builder notation with Java, like { x \* 2 | x E {1,2,3,4} ^ x is even }. Now in use in the jComprehension java library.

Full article: List%20Comprehensions%20in%20Java.pdf

ABSTRACT

In many programming languages, such as Haskell or python, it is popular to write list comprehensions, like we write in algebra

as the list of all even numbers in the set of real numbers. This article proposes an implementation of list comprehensions in Java, and then provides the implementation of the functions map and filter using the proposed code. Also, it supports binding more than one variable to the list’s definition, so we will be able to implement lists of cartesian products and also to add a relation that must hold between the two variables, like

INTRODUCTION

We may remember from algebra that we can define lists as lists comprehensions, which uses a special notation called set-builder notation. For example,

denotes the set of all pairs (x,y) such that x and y are real numbers and the product of both numbers equals the sum.

Today, this syntactic construct is integrated in many programming languages, such as

Haskell

[x \* 2 | x <- [0 .. 99], x \* x > 3]

python

S = [2\*x for x in range(100) if x\*\*2 > 3]

C#

var ns = Enumerable.Range(0, 100) .Where(x => x\*x > 3) .Select(x => x\*2);

Scala

val s = for (x <- Stream.from(0); if x\*x > 3) yield 2\*x

Ruby

(0..100).select { |x| x\*\*2 > 3 } .map { |x| 2\*x }

and more. But the Java language does not provide a syntactic construct for this concept.

The usage of list comprehensions in Haskell motivated this article. Bringing the set-builder notation to Java, this different way of thinking problems. Instead of using for or while statements in your programs, you may work with map and filter functions, but take into account that thinking problems as list comprehensions is also another good way to practice programming.

The structure of this article follows:

1. Set-builder notation and list comprehensions definitions as we saw in algebra.

2. Some words about the implementation of Haskell’s list comprehensions. The implementation of map and filter with list comprehensions in Haskell.

3. The proposed implementation of list comprehensions in Java. The implementation of map, filter and the cartesian product with our new list comprehensions.

4. Discussion. Is this useful at industrial level? Possible enhancements and future work.

SET-BUILDER NOTATION

In set theory there is a popular way of defining sets: set-builder notation. Any list defined in this way is called a list comprehension. For example, let’s define the set of all integers that are even and are bigger or equals than 5

Which is read as give me the set of all x such that x belongs to the set of the integer numbers, x is even and x is greater than 5.

Going formal, a set is composed by three sections: a variable, a colon or vertical bar separator, and a logical predicate, which are contained in curly brackets. We may also quantify variables, either by using the existential quantifier, like we use in this definition of the set of all natural even numbers

or the universal quantifier, negations, etc.

Sometimes, we have to decide if we want our set-builder notation to be able to quantify not only elements that belong to a set, but also under sets as well. If we do this, we may encounter some situations like the one described as the Russell’s Paradox

which is read as give me the set of all sets S such that S does not contain themselves.

Let’s write the functions map and filter with the set-builder notation we learned. For the first one, we want the set of all the elements that are in a set S but with some transformation applied, let’s say the function f which holds

We could write our version of map like this(with some syntax sugar on the variable section)

And now we can do the same for the filter function

where x must belong to the set S and hold the predicate

LIST COMPREHENSIONS IN HASKELL

Haskell is one of the programming languages that supports writing list comprehensions. Let’s consider the example

[toUpper c | c <- s]

where s is a string such as “Hello” (s :: String). Strings in Haskell are lists of characters; the generator c <- s feeds each character of s in turn to the left-hand expression toUpper c, building a new list. The result of this list comprehension is “HELLO”.

And for multiple generators, we may have

[(i,j) | i <- [1,2], j <- [1..4]]

yielding the result

[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2), (2,3),(2,4)]

In Haskell, list comprehensions are translated into equivalent definitions in terms of map, and concat. The translation rules are

[e |True] = [e]

[e | q] = [e | q, True]

[e | b, Q] = if b then [e | Q] else []

[e | p <- xs, Q] = let ok p = [e | Q] ok \_ = [] in concat (map ok xs)

Now, we can write the functions map and filter as a list comprehensions as we did on the previous section

map f xs = [f x | x <- xs]

filter p xs = [x | x <- xs, p x]

LIST COMPREHENSIONS IN JAVA

Here is the proposed solution for list comprehensions in Java, where a new class ListComprehension has been added.

would be now expressed in Java as

One of the main goals when defining this Java API is that we want the user to read this code and read the same concepts as in the set-builder algebraic notation: give me the set (or list comprehension) of all x such that x belongs to the list of [1,2,3,4] and x is even.

DISCUSSION

We got able to implement the functions map and filter using list comprehensions written in set-builder notation, Haskell and Java code, the latest one with an implementation of this article. We were also able to bind more than one variable, and write the cartesian product of two sets of integers and add a relation between the variables.

But our new implementation lacks of some functionality though, such as quantifiers exists and for-all under elements, quantifiers under sets (which we do have in set-builder notation), among others. Could it be possible to express quantification under sets in the Java implementation? How the Russells’s Paradox case would behave?

Another thing to take into consideration is the usage of list comprehensions in industrial production code. We have seen how Java adopted some functional programming fundamentals in its newer releases (Java 8), such as lambdas, map, filter, etc; may be is not too crazy to think that Java will adopt list comprehensions at some point, as Scala, Clojure and other programming languages have done.

For-Comprehension Free Monads in Scala

A free monad is a construction that allows you to build a monad from any functor. There exists rich literature that explains this concept, notably from Cats and Scalaz documents.

In Scala, free monad allows us to model a workflow using for-comprehension through its monadic operations; it lifts the operations to free monads and is run by an interpreter. However, Scala’s for-comprehension has more to offer than monadic operations.

In this article, we introduce our enhancement by extending free monads beyond monadic operations. We show that this addition makes the free monads more powerful and provides more elegant encoding, without breaking monad law. We briefly explain the building blocks of free monad without making elaborating on its details. Readers are encouraged to understand its full context through online resources. We illustrate our enhancement based on the sample implementation in this article: Free Monads Explained (Pt. 1), without using any of the Scala libraries.

Basic Form of Free Monads

Free monads consist of six parts.

The central piece is a trait Free[F[\_], A], with monadic operations. This is the container that encapsulates the user operations.

sealed trait Free[F[\_], A] {

def flatMap[B](f: A => Free[F, B]): Free[F, B] = this match {

case Return(a) => f(a)

case FlatMap(sub, cont) => {

FlatMap(sub, cont andThen (\_ flatMap f))

}

}

def map[B](f: A => B): Free[F, B] = flatMap(a => Return(f(a)))

}

case class Return[F[\_], A](a: A) extends Free[F, A]

case class FlatMap[F[\_], I, A](sub: F[I], cont: I => Free[F, A]) extends Free[F, A]

User Instructions, for instance:

sealed trait AskTell[A]

case class Ask(message:String) extends AskTell[String]

case class Tell(message:String) extends AskTell[String]

An implicit function that lifts the user instructions to free monads.

implicit def liftF[F[\_], A](fa: F[A]): Free[F, A] = FlatMap(fa, Return.apply)

An interpreter that runs the free monads.

def runFree[F[\_], A](prg: Free[F, A], executor: Executor[F]): A = {

prg match {

case Return(a) => a

case FlatMap(sub, cont, filter) => {

runFree(cont(executor.exec(sub, filter)), executor)

}

}

}

Implementations of user instructions encapsulated in an executor:

sealed trait Executor[F[\_]] {

def exec[A](fa: F[A]): A

}

val asktellExec = new Executor[AskTell] {

override def exec[A](fa: AskTell[A]):A = fa match {

case Ask(message) => {

println(message)

val name = scala.io.StdIn.readLine()

name.asInstanceOf[A]

}

case Tell(message) => {

println(message)

message.asInstanceOf[A]

}

}

}

}

Workflow modeling

val asktell = for {

name <- Ask("What is your name?")

\_ <- Tell(s"Hello ${name}!")

} yield ()

And finally, run the workflow:

runFree(asktell, asktellExec)

For-Comprehension in Scala

For-comprehension is a lightweight notation for expressing sequence comprehensions. It takes the following forms:

For, with yield, is translated into map:

for(x <- List(1,2,3)) yield x\*x

Which will be translated to:

List(1,2,3).map {x => x\*x}

Then, nested for with yield is translated to flatMap and map:

for(x <- List(1,2,3); y <- List(4,5,6)) yield x\*y

Which will be translated to:

List(1,2,3).flatMap { x =>

List(4,5,6).map { y => x\*y}

}

For, without yield, is translated to foreach, anything inside foreach are actions with a side effect, for instance, sending emails, writing to the database, etc.

for(x <- List(1,2,3)) println(x)

Which will be translated to:

List(1,2,3).foreach{x => println(x)}

For, with guard, is translated to withFilter:

for(x <- List(1,2,3) if x ==3 ) yield x\*x

Which is translated to:

List(1,2,3).withFilter(x == 3)..map {x => x\*x}

As we can see, free monad provides map and flatMap; it covers the first and second case in for-comprehension.

Enhanced Free Monads

Free monads provide a natural transformation of workflow through monadic operations; however, the workflow is not only an independent sequence of user instructions; actions with side effects can be part of the workflow, and the workflow sequence can be branched out depending on predicates. As discussed, the current free monad implementation cannot utilize the full potential of for-comprehension. The missing parts are foreach translation and filters, which will be addressed in our enhanced free monad.

Here are the encodings to each part of the enhanced free monad.

Free Monad definition:

def foreach[U](f: A => U) : Free[F, A] = this match {

case Return(a) => {

f(a) match {

case free : Free[F, A] => free

case \_ => this

}

}

case FlatMap(sub, cont, filter ) => {

FlatMap(sub, cont andThen (\_ foreach f), filter)

}

}

def withFilter(f: A => Boolean): Free[F, A] = this match {

case Return(a) => this

case FlatMap(sub, cont, filter ) => FlatMap(sub, cont, f(ph))

}

The key addition to an enhanced free monad is the presence of foreach and withFilter for for-comprehension to access. They cover the third and fourth case of for-comprehension discussed in the last section. A key change to FlatMap is that it carries the evaluation of the predicate in its construction and carried on for future execution:

case class FlatMap[F[\_], I, A](sub: F[I], cont: I => Free[F, A], filter:Boolean) extends Free[F, A]

The user instruction interface remains the same

Implicit lift

implicit def liftF[F[\_], A](fa: F[A]): Free[F, A] = FlatMap(fa, Return.apply, true)

Implicit lift function adds a Boolean guard with a default value of true.

The interpreter:

def runFree[F[\_], A](prg: Free[F, A], executor: Executor[F]): A = {

prg match {

case Return(a) => a

case FlatMap(sub, cont, filter) => {

runFree(cont(executor.exec(sub, filter)), executor)

}

}

}

The interpreter constructs the calling sequence and passes the guard recursively.

User instruction implementation

val asktellExec = new Executor[AskTell] {

override def exec[A](fa: AskTell[A], filter:Boolean) = fa match {

case Ask(message) if filter => {

println(message)

val result = scala.io.StdIn.readLine()

result.asInstanceOf[A]

}

case Tell(message) if filter => {

println(message)

message.asInstanceOf[A]

}

case \_ => "filtered out".asInstanceOf[A]

}

The guard is applied here before execution.

Workflow modeling:

val asktell = for {

hour <- Ask("What time is it?")

\_ <- Tell(s"Good Morning, it is ${hour}am") if (hour.toInt <= 12)

\_ <- Tell(s"Good afternoon, it is ${hour.toInt -12}pm") if (hour.toInt > 12)

} yield ()

​

val asktell2 = for {

firstname <- Ask("what is your first name?")

lastname <- Ask("what is your last name?")

age <- Ask("what is your age?")

} {

println(s"${firstname} ${lastname}, ${age} years old")

}

We notice that we are able to use predicates, as well as for-comprehension with actions.

The execution of the workflow remains the same:

runFree(asktell, asktellExec)

The complete implementation can be found on GitHub.

Conclusion

In this article, we summarized the building blocks of a free monad, as well as explained for-comprehension in Scala. We have shown how free monads work with for-comprehension, identified the areas to be improved, and provided the implementation.

The enhanced free monad implementation maintains the properties of free monads – stack-free and natural transformation. By making free monads fully for-comprehension, the encoding gained firepower to model more dynamic operations; the interpreter separates the concerns from execution; and the client program remains the same.

Free Monads Explained (pt 1)

Building composable DSLs

Free monad?

The core idea of Free is to switch from effectful functions (that might be impure) to plain data structures that represents our domain logic. Those data structures are composed in a structure that is free to interpretation and the exact implementation of our program can be decided later.

Imperative example

As a use case example we will take a program that asks for user’s name a greets him:

println("Greetings!")

println("What is your name?")

val name = scala.io.StdIn.readLine()

println(s"Hi $name!")

As usual, we will go through series of small transformations trying to generalize things and eventually come up with Free monad implementation.

Creating an algebra

First off, lets define an algebra that represents our program. There are two clear and distinct operations:

Tell represents an action to tell something to a user. We don’t say how, it might be a standard out print, message on a screen, etc.

Ask questions user for something and returns the answer.

sealed trait UserInteraction[A]

case class Tell(statement: String) extends UserInteraction[Unit]

case class Ask(question: String) extends UserInteraction[Unit]

The imperative program from above can be described as a List of Asks and Tells:

val program = List(

Tell("Greetings!"),

Ask("What is your name?"),

Tell("Hi, nice to meet you!"))

Instead of calling functions we construct a description of a program. In order to execute such program we need to know how to “execute” each individual UserInteraction and then simply map it over the list:

def execute[A](ui: UserInteraction[A]): A = ui match {

case Tell(statement) =>

println(statement)

case Ask(question) =>

println(question)

val answer = scala.io.StdIn.readLine()

() // ignoring the answer for now

}

def run[A](program: List[UserInteraction[A]]): Unit = program.foreach(execute)

The interpreter (execute + run) is an implementation of our program description, this is the place where all mutation and side effects can happen.

By the way, our new program is slightly different from the first imperative program — instead of greeting user by his name it just says generic “nice to meet you”. Our UserInteraction data structure is not very useful right now, we don’t have a way of getting the values of previous Ask's and referring them in Tell. The steps are sequential and depend on the value of a previous computation. Sounds like a Monad, right? If our UserInteraction would be a Monad (and lets even say we already implemented pure and flatMap) that would allow us to rewrite our program like this:

val program = for {

\_ <- Tell("Greetings!")

name <- Ask("What is your name?")

\_ <- Tell("Hi $name!")

} yield ()

Now we can compose our user actions and access computation results. The problem is that program not a List of instructions anymore:

def run[A](program: UserInteraction[A]): Unit = ??? // match on program?

By introducing monadic bind we lost the ability to introspect the data structure, there is no way of knowing what our resulting UserAction[A] was made of and interpret each step.

Monadception

First, let’s change our Ask to ‘return’ a value of a proper type in order to capture it in the monadic bind:

sealed trait UserInteraction[A]

// An effect that takes a String and retuns Unit vvvv

case class Tell(statement: String) extends UserInteraction[Unit]

// An effect that takes a String and retuns a String vvvvvv

case class Ask(question: String) extends UserInteraction[String]

We know that UserInteraction needs to be monadic, be able to compose sequential computations but at the same time preserve information about the computation steps. The flatMap operation takes an F[A], a function A => F[B] and returns an F[B], this is where information is lost, same as with regular functions composition we lose information about what were the original composed functions. What if we could go meta again and replace calls to flatMap with a data structure? Eventually I want to get something like this:

FlatMap(Tell("Greetings!"), (\_) =>

FlatMap(Ask("What is your name?"), (name) =>

Return(Tell(s"Hi $name!"))))

By introducing FlatMap and Return we captured what it’s like for something to be a Monad and this is what FreeMonad is:

sealed trait Free[F[\_], A]

case class Return[F[\_], A](a: A) extends Free[F, A]

case class FlatMap[F[\_], I, A](sub: F[I], cont: I => Free[F, A]) extends Free[F, A]

Imagine the F[\_] being our UserInteraction but it can be any type constructor and there are no constraints on the F being a Monad or something. Return and FlatMap are the analogy of pure and flatMap on the monad — putting a value in a context and gluing computation together.

Free is a recursive structure where each subsequent computation can access the result of a previous computation. This is all we need to build composable programs using plain data structures that are free to interpretation. Let’s return to this example:

FlatMap(Tell("Greetings!"), (\_) =>

FlatMap(Ask("What is your name?"), (name) =>

Return(Tell(s"Hi $name!"))))

We want to end up having this kind of structure but it would be awkward to write programs by directly creating data structures. Besides, scala has a nice syntax to make it easier — “for comprehension”. This is how I’d like to see our program:

val program = for {

\_ <- Tell("Greetings!")

name <- Ask("What is your name?")

\_ <- Tell("Hi $name!")

} yield ()

To make that work we need two things:

Free has to be a Monad (compiler will look for flatMap and map defined on Free trait)

A way to construct values of Free[UserInteraction, A]. We want flatMap to be called on Free, not on UserInteraction (which is not a Monad and that’s the point)

Free as a Monad

“For comprehension” is desugared into a sequence of flatMap calls ending up with map. They have to be defined on the Free trait:

sealed trait Free[F[\_], A] {

def flatMap[B](f: A => Free[F, B]): Free[F, B] = this match {

case Return(a) => f(a)

case FlatMap(sub, cont) => FlatMap(sub, cont andThen (\_ flatMap f))

}

def map[B](f: A => B): Free[F, B] = flatMap(a => Return(f(a)))

}

flatMap has to deal with two cases: Return is simply about applying the f to the inner a, FlatMap is about creating the same FlatMap whose continuation created by composing self cont with the result of flatMaping f. Sounds complicated, sometimes its just better to follow the types and try to implement it yourself until it “clicks”.

Now we can do something like this:

// say we have

// firstProgram: Free[UserInteraction, String]

// secondProgram: Free[UserInteraction, String]

val program = for {

\_ <- firstProgram

\_ <- secondProgram

} yield ()

Creating Free stuff

How do we actually construct a Free[UserInteraction, A]? There has to be a function:

toFree[A](fa: UserInteraction[A]): Free[UserInteraction, A] = FlatMap(fa, Return.apply);

toFree takes a Tell or Ask and returns a free monad constructed by FlatMaping the value with a function that will just yield the result. That allows us to do the following:

val program = for {

\_ <- toFree(Tell("Hello!"))

name <- toFree(Ask("What is your name?"))

\_ <- toFree(Tell(s"Hi, $name"))

} yield ()

There is a trick to make the syntax nicer and not having to write toFree every time — make toFree available for implicit conversions:

// vvvvv

implicit def toFree[A](fa: UserInteraction[A]): Free[UserInteraction, A] =

FlatMap(fa, Return.apply)

val program = for {

\_ <- Tell("Hello!")

name <- Ask("What is your name?")

\_ <- Tell(s"Hi, $name")

} yield ()

Now thanks to Free being a Monad and having a way to implicitly create Free from Tell and Ask our program looks exactly how we wanted to write it in the first place.

Do you even lift, Free?

Looking at toFree signature we can spot an improvement — abstracting over higher order type constructor it can be written in a more generic way, not specific to UserInteraction:

// also liftF is a usual name for creating Free

implicit def liftF[F[\_], A](fa: F[A]): Free[F, A] = FlatMap(fa, Return.apply)

The problem here is that now it matches concrete type constructors:

// vvvv Tell, but we want UserInteraction

val ua: Free[Tell, String] = liftF(Tell("Hello"))

We would run into compilation errors trying to compile for comprehension. The workaround is quite simple:

Define a type alias fixing UserInteraction to be a generator for Free which will represent the program’s DSL

Create functions to construct DSL by lifting our algebra types

// generalized liftF function

implicit def liftF[F[\_], A](fa: F[A]): Free[F, A] = FlatMap(fa, Return.apply)

// Algebra type

type InteractionDsl[A] = Free[UserInteraction, A]

// "Smart" constructors

def tell[A](str: A): InteractionDsl[A] = liftF(Tell(str))

def ask[A](answer: A): InteractionDsl[A] = liftF(Ask(answer))

// Usage

val program: Free[UserInteraction, Unit] = for {

\_ <- tell("Hello!")

name <- ask("What is your name?")

\_ <- tell(s"Hi, $name")

} yield ()

Running the program

So now we have a nice syntax and data structure representing our program, how do we actually run it? Similar to when we had just a List of instructions we can fold the Free executing each instruction. So currently our goal is: given program Free[F, A] traverse the Free structure evaluating each step and threading results to subsequent computations.

But first, lets start off simple with implementing runInteractionDSL for our concrete program InteractionDsl[A] and generalize to a Free[F, A] later. The first obvious thing to do is to pattern match on the argument:

def runInteractionDSL[A](prg: InteractionDsl[A]): A = prg match {

case Return(a) => a

case FlatMap(sub, cont) => ??? // how to execute sub?

}

How to run sub which is UserInteraction[A]? This can be either Tell[A] or Ask[A], both of them has different meaning and there is no direct way to run them. We can use our previously implemented execute function to run sub operations, getting result back, feeding it back to cont and recursively call runInteractionDSL:

// Execution can be impure, side effects are OK here

def execute[A](ui: UserInteraction[A]): A = ui match {

case Tell(str) =>

println(str)

str

case Ask(question) =>

println(question)

val answer = scala.io.StdIn.readLine()

answer.asInstanceOf[A]

}

def runInteractionDSL[A](prg: InteractionDsl[A]): A = prg match {

case Return(a) => a

case FlatMap(sub, cont) => {

val result = execute(sub)

runInteractionDSL(cont(result))

}

}

runInteractionDSL(program)

Generalizing over the algebra

Lets make a step towards more generic implementation. The runFree should be able to execute any Free[F, A], not just InteractionDsl[A](e.g. Free[UserInteraction, A]).

// Won't compile

def runFree[F[\_], A](prg: Free[F, A]): A = prg match {

case Return(a) => a

case FlatMap(sub, cont) => {

val result = execute(sub) // <-- a problem

runFree(cont(result))

}

}

The execute function is specific to UserInteraction and it’s not applicable to Free[F, A]. It has to be a generic function that by given F[A] will give us A back and possibly do some effects. Also, it has to be provided by the user and passed into runFree:

sealed trait Executor[F[\_]] {

def exec[A](fa: F[A]): A

}

def runFree[F[\_], A](prg: Free[F, A], executor: Executor[F]): A = prg match {

case Return(a) => a

case FlatMap(sub, cont) => {

val result = executor.exec(sub)

runFree(cont(result), executor)

}

}

// Usage

val consoleExec = new Executor[UserInteraction] {

override def exec[A](fa: UserInteraction[A]) = fa match {

case Tell(str) =>

println(str)

str

case Ask(question) =>

println(question)

val answer = scala.io.StdIn.readLine()

answer.asInstanceOf[A]

}

}

runFree(program, consoleExec)

By the way, Executor is not something new but just a Comonad

That’s pretty much it, we can run any Free[F, A] by providing a program to run and a custom “executor” that interprets our algebra. Looks great, but we can do even better.

Natural transformation:

There is one more thing that can be generalized and that our “executor” function. Looking at the signature F[A] => A is actually a special case of functor transformation F[A] => Id[A] where Id is just a type alias for A (type Id[A] = A). This kind of transformation between functors is called “natural transformation”:

sealed trait NaturalTransformation[F[\_], G[\_]] {

def transform[A](fa: F[A]): G[A] // <-- G[A] instead of just A

}

Natural transformation is just a function that maps values in one context to another (F[A] ~> G[A]). In our case it will map UserInteraction data types to an Id:

sealed trait NaturalTransformation[F[\_], G[\_]] {

def transform[A](fa: F[A]): G[A]

}

// Won't compile

def runFree[F[\_], G[\_], A](prg: Free[F, A], nt: NaturalTransformation[F, G]): A = prg match {

case Return(a) => a

case FlatMap(sub, cont) => {

val transformed = nt.transform(sub)

runFree(cont(transformed), nt) // <- hmmm

}

}

Previously Executor returned A which we could directly pass into continuation but now it returns G[A]. We can resolve it by requiring G to be a Monad and use flatMap to thread inner A into cont and return G[A]. The return value of runFree has to be G[A] instead of A as well:

// We could define Monad trait and monad instance for Id ourselves

// but since it's not critical for this article and for keeping

// things simple let's just use Cats

import cats.{Id, Monad}

sealed trait NaturalTransformation[F[\_], G[\_]] {

def transform[A](fa: F[A]): G[A]

}

def runFree[F[\_], G[\_], A]

(prg: Free[F, A], nt: NaturalTransformation[F, G])

(implicit M: Monad[G]): G[A] =

prg match {

case Return(a) => Monad[G].pure(a)

case FlatMap(sub, cont) => {

val transformed = nt.transform(sub)

Monad[G].flatMap(transformed) { a => runFree(cont(a), nt) }

}

}

val consoleIO = new NaturalTransformation[UserInteraction, Id] {

override def transform[A](fa: UserInteraction[A]) = fa match {

case Tell(str) =>

println(str)

str

case Ask(question) =>

println(question)

val answer = scala.io.StdIn.readLine()

answer.asInstanceOf[A]

}

}

runFree(program, consoleIO)

Yay, finally we’re done. The real implementation that you might find in Cats is a bit different but most of that is about syntax and style — everything regarding Free is defined within the trait, NaturalTransformation is calledFunctionK, transform is apply, run is foldMap with curried arguments. The idea stays the same and hopefully our step by step approach served as a relatively simple introduction to the Free concept.

DSLs with the Free Monad in Java 8 : Part I

A program in Free

Recently, there has been an explosion of interest in the Scala community about a data structure known as Free or the Free Monad. It allows Scala developers to define internal mini languages, and hence programs, inside their own applications that they can interpret and execute independently as needed. Free behaves much like Stream and Optional do in Java 8. While the map and flatMap methods on Stream capture the transformation steps to be applied to a Stream of data, the map and flatMap methods on Free can capture the sequencing of commands to be executed in a custom program.

This technique, of defining DSLs can be used to enforce SOLID design principles (in particular Dependency Inversion) and allows us to ensure that we don’t mix abstraction levels at critical junctures in our code. While our DSLs lack the syntax for lower or higher levels of abstraction, they force us to keep to the right path. We can also attach different interpreters to our DSLs at runtime, so for example in our unit tests the interpreter can mock the live service calls we’d actually make in a production environment.

While Free itself is an advanced concept, defining DSLs in terms in of Free can be done entirely in terms of pure Java 8 and is arguably a simpler task than defining and combining multiple grammars and parsers using tooling such as Antlr. However I suspect most Java developers will find this article useful as a primer on a concept that may seem alien and even unintelligible when expressed in other languages.

The examples in Part I will be based on a simplified implementation of Free provided in cyclops-react called Unrestricted.

This series of articles / tutorials is based on the Haskell Blog Post by Gabriel Gonzalez called Why Free Monads Matter, and its Java Conversion by Kenji Yoshida.

The article continues below, where we will

define a simple DSL

a program written in that DSL

2 Interpreters for our new DSL that handle program execution differently

Expressing Free in Java

Let’s start by defining a simple program

output('A');

bell();

output('B');

done();

Open in app

Get started

ModernNerd Code

DSLs with the Free Monad in Java 8 : Part I

A program in Free

Recently, there has been an explosion of interest in the Scala community about a data structure known as Free or the Free Monad. It allows Scala developers to define internal mini languages, and hence programs, inside their own applications that they can interpret and execute independently as needed. Free behaves much like Stream and Optional do in Java 8. While the map and flatMap methods on Stream capture the transformation steps to be applied to a Stream of data, the map and flatMap methods on Free can capture the sequencing of commands to be executed in a custom program.

This technique, of defining DSLs can be used to enforce SOLID design principles (in particular Dependency Inversion) and allows us to ensure that we don’t mix abstraction levels at critical junctures in our code. While our DSLs lack the syntax for lower or higher levels of abstraction, they force us to keep to the right path. We can also attach different interpreters to our DSLs at runtime, so for example in our unit tests the interpreter can mock the live service calls we’d actually make in a production environment.

While Free itself is an advanced concept, defining DSLs in terms in of Free can be done entirely in terms of pure Java 8 and is arguably a simpler task than defining and combining multiple grammars and parsers using tooling such as Antlr. However I suspect most Java developers will find this article useful as a primer on a concept that may seem alien and even unintelligible when expressed in other languages.

The examples in Part I will be based on a simplified implementation of Free provided in cyclops-react called Unrestricted.

This series of articles / tutorials is based on the Haskell Blog Post by Gabriel Gonzalez called Why Free Monads Matter, and its Java Conversion by Kenji Yoshida.

The article continues below, where we will

define a simple DSL

a program written in that DSL

2 Interpreters for our new DSL that handle program execution differently

Expressing Free in Java

Let’s start by defining a simple program

In this program we want to

output A to the console

ring a bell

output B to the console

terminate

Using the Unrestricted type in cyclops-react (a simplified implementation of Free) we can express and capture this program in pure Java as follows :-

import static cyclops.control.Unrestricted.comprehensions.forEach;

Unrestricted<Void> program = forEach( output('A'),

\_\_ -> bell(),

\_\_ -> output('B'),

\_\_ -> done());

The meat and bones of the program are on the right, and boiler plate to capture this is defined on the left (don’t worry we will explain what is going on here soon).

The variable ‘program’ captures the sequence of commands to be executed. Once the above code is run, we end up with a linked Data Structure that could be visualized something like this;

The returned Unrestricted (free monad) Instance has an embedded reference to the first command in our program which in turn can generate a reference to the next Unrestricted instance and so on. To execute our program all we need do is walk this data structure (using the resume method on Unrestricted) and attach an interpreter to handle each of the commands.

Building an Interpreter

In our interpreter we want to walk the captured data structure and interpret each of the commands. At this stage we know we will have to handle output, bell and done commands. We will be able to put more meat on the bones of our interpreter after introducing a few more concepts.

public class StringBuildingInterpreter{

public static <R> String interpret(Unrestricted<R> program){

//walk the Free data structure and handle each command,

//by delegating to the appropriate method

}

public static String handleOutput(Output<Unrestricted<Command<Void>> output){

//here we should handle the output command and because the output

//command will contain the link

//to the next stage in the program (note the generic type signature)

//we should recursively call interpret

}

public static String handleBell(Bell<Unrestricted<Command<Void>> bell){

//here we should handle the Bell command and because the bell command

//will contain the link

//to the next stage in the program (note the generic type signature)

//we should recursively call interpret

}

public static String String handleDone(Done<Void> done){

//handling done is simpler as it terminates the program,

//we have no more interpretation to do

return “done\n”\*\*;

}

}

Free works with Functors

Java 8 introduced a number of Functors (or data types that can transformed via a function) including Stream, Optional and CompletableFuture.

When we make use of the map method on a Stream, we are treating it as a Functor.

Stream.of(1,2,3)

.map(i->i+1);

We can also define our own Functors, for example to make List behave as a Functor we could define a map method for Lists.

public <T,R> List<R> map(Function<? super T, ? extends R> mapper);

public interface Functor<T>{

<R> Functor<R> map(Function<? super T, ? extends R> mapper);

}

In cyclops-react the name Functor is reserved for the more abstract Functor type class that leverages psuedo Higher Kinded Types (of which more in part II of this series). The type Transformable is used to define a simpler / easier to use Functor interface -> Link To Transformable class

Commands are Transformable

We can define each of our programs’ commands by defining an appropriate map method. Our first step is to create an appropriate base class for our DSLs’ commands.

abstract class Command<T> implements Transformable<T>{

public abstract <R> Command<R> map(Function<? super T, ? extends R> f);

}

Our map method will be used by our Free implementation (Unrestricted) to realize the chain of Unrestricted instances that represent our program. The provided Function will often return an instance of Unrestricted that can lead us to the next Command in the chain (and so on).

Done

This makes defining Done really simple, as Done represents the last stage of the program we can pretty much ignore the supplied Function.

public class Done<T> extends Command<T> {

@Override

public <R> Done<R> map(Function<? super T, ? extends R> f) {

return new Done<>();

}

}

Bell

Bell is a little more complex.

We need to define a generic variable that can represent a path to the next stage (although it is not shown in this example we can also use next to pass data between commands). In practice, the next stage of the program will stored inside Bell as an instance of Unrestricted / Free.

In the map method we need to apply the provided function to our next variable.

public class Bell<T> extends Command<T> {

private final T next;

private Bell(final T next) {

this.next = next;

}

@Override

public <R> Bell<R> map(final Function<? super T,? extends R> f) {

return new Bell<>(f.apply(next));

}

}

Output

The implementation of Output is pretty much identical to that of Bell, except that we also need to capture & store the character that we wish to output to the console.

public class Output<T> extends Command<T>{

private final char c;

private final T next;

private Output(final char c, final T next) {

this.c = c;

this.next = next;

}

@Override

public <R> Output<R> map(final Function<? super T, ? extends R> f) {

return new Output<>(c, f.apply(next));

}

}

Commands want to be Free

In our program all our commands are embedded inside instances of Free / Unrestricted. We use a static method to create each command.

public static Unrestricted<Command<Void>> output(final char out){

return Unrestricted.liftF(new Output<>(out, null));

}

To embed an Output instance inside an instance of Free we can create a static method to do that and make use of the ‘LiftF’ method on Unrestricted.

public static Unrestricted<Command<Void> bell(){

return Unrestricted.liftF(new Bell<Void>(null));

}

public static Unrestricted<Command<Void> done(){

return Unrestricted.liftF(new Done<Void>());

}

Open in app

Get started

ModernNerd Code

DSLs with the Free Monad in Java 8 : Part I

John McClean

John McClean

May 30, 2017 · 8 min read

A program in Free

Recently, there has been an explosion of interest in the Scala community about a data structure known as Free or the Free Monad. It allows Scala developers to define internal mini languages, and hence programs, inside their own applications that they can interpret and execute independently as needed. Free behaves much like Stream and Optional do in Java 8. While the map and flatMap methods on Stream capture the transformation steps to be applied to a Stream of data, the map and flatMap methods on Free can capture the sequencing of commands to be executed in a custom program.

This technique, of defining DSLs can be used to enforce SOLID design principles (in particular Dependency Inversion) and allows us to ensure that we don’t mix abstraction levels at critical junctures in our code. While our DSLs lack the syntax for lower or higher levels of abstraction, they force us to keep to the right path. We can also attach different interpreters to our DSLs at runtime, so for example in our unit tests the interpreter can mock the live service calls we’d actually make in a production environment.

While Free itself is an advanced concept, defining DSLs in terms in of Free can be done entirely in terms of pure Java 8 and is arguably a simpler task than defining and combining multiple grammars and parsers using tooling such as Antlr. However I suspect most Java developers will find this article useful as a primer on a concept that may seem alien and even unintelligible when expressed in other languages.

The examples in Part I will be based on a simplified implementation of Free provided in cyclops-react called Unrestricted.

This series of articles / tutorials is based on the Haskell Blog Post by Gabriel Gonzalez called Why Free Monads Matter, and its Java Conversion by Kenji Yoshida.

The article continues below, where we will

define a simple DSL

a program written in that DSL

2 Interpreters for our new DSL that handle program execution differently

Expressing Free in Java

Let’s start by defining a simple program

In this program we want to

output A to the console

ring a bell

output B to the console

terminate

Using the Unrestricted type in cyclops-react (a simplified implementation of Free) we can express and capture this program in pure Java as follows :-

The meat and bones of the program are on the right, and boiler plate to capture this is defined on the left (don’t worry we will explain what is going on here soon).

The variable ‘program’ captures the sequence of commands to be executed. Once the above code is run, we end up with a linked Data Structure that could be visualized something like this;

The returned Unrestricted (free monad) Instance has an embedded reference to the first command in our program which in turn can generate a reference to the next Unrestricted instance and so on. To execute our program all we need do is walk this data structure (using the resume method on Unrestricted) and attach an interpreter to handle each of the commands.

Building an Interpreter

In our interpreter we want to walk the captured data structure and interpret each of the commands. At this stage we know we will have to handle output, bell and done commands. We will be able to put more meat on the bones of our interpreter after introducing a few more concepts.

Free works with Functors

Java 8 introduced a number of Functors (or data types that can transformed via a function) including Stream, Optional and CompletableFuture.

When we make use of the map method on a Stream, we are treating it as a Functor.

We can also define our own Functors, for example to make List behave as a Functor we could define a map method for Lists.

A more general Functor interface could be defined something like this

In cyclops-react the name Functor is reserved for the more abstract Functor type class that leverages psuedo Higher Kinded Types (of which more in part II of this series). The type Transformable is used to define a simpler / easier to use Functor interface -> Link To Transformable class

Commands are Transformable

We can define each of our programs’ commands by defining an appropriate map method. Our first step is to create an appropriate base class for our DSLs’ commands.

Our map method will be used by our Free implementation (Unrestricted) to realize the chain of Unrestricted instances that represent our program. The provided Function will often return an instance of Unrestricted that can lead us to the next Command in the chain (and so on).

Done

This makes defining Done really simple, as Done represents the last stage of the program we can pretty much ignore the supplied Function.

Bell

Bell is a little more complex.

We need to define a generic variable that can represent a path to the next stage (although it is not shown in this example we can also use next to pass data between commands). In practice, the next stage of the program will stored inside Bell as an instance of Unrestricted / Free.

In the map method we need to apply the provided function to our next variable.

Output

The implementation of Output is pretty much identical to that of Bell, except that we also need to capture & store the character that we wish to output to the console.

Commands want to be Free

In our program all our commands are embedded inside instances of Free / Unrestricted. We use a static method to create each command.

To embed an Output instance inside an instance of Free we can create a static method to do that and make use of the ‘LiftF’ method on Unrestricted.

LiftF creates an instance of Unrestricted with an embedded functor (an instance of Transformable). As all our Commands implement Transformable we can pass them to LiftF.

Interpreting our Java Program

Hopefully the boilerplate makes a little more sense now. On the right we are creating instances of our commands embedded inside Free / Unrestricted.

On the left we are defining our program in terms of a Free instance. On the right each command is provided as a parameter to the forEach method — either directly or as the return value from a function.

Ultimately the variable program represents the sequencing of these commands in a data-structure something like a linked chain of Free / Unrestricted instances. Any combination of the commands we have defined can be put together in this way to define any acceptable program. We just need to finish our interpreter to be able to execute it.

Interpreting the Output Command

When we have an output command we want to do 2 things.

We want to append the character defined as an operand to this Command to our output String

We want to interpret the next step of the program

With Java 8 lambdas we can implement the visitor pattern (aka pattern matching in functional languages) for this type really simply. In our visitor we should provide both the captured character and the next step of the program to a user defined function.

public class Output<T> extends Command<T>{

private final char c;

private final T next;

private Output(final char c, final T next) {

this.c = c;

this.next = next;

}

public <R> R visit(final BiFunction<Character, ? super T,? extends R> charAndNextStageFn) {

return charAndNextStageFn.apply(c, next);

}

//.. more code //

}

We can then make use of this visit method when we handle output by passing it a BiFunction that appends the appropriate character to an output String and recursively interprets the next stage of the program.

public class StringBuildingInterpreter {

public static <R> String handleOutput(Output<Unrestricted<Command<R>> output){

//here we should handle the output command and because the output

//command will contain the link

//to the next stage in the program (note the generic type signature)

//we should recursively call interpret

return output.visit((c, next) -> "emitted " + c + "\n" + interpret(next));

}

public static <R> String interpret(Unrestricted<R> program){

//walk the Free data structure and handle each command,

//by delegating to the appropriate method

}

//..more code//

}

Get started

ModernNerd Code

DSLs with the Free Monad in Java 8 : Part I

John McClean

John McClean

May 30, 2017 · 8 min read

A program in Free

Recently, there has been an explosion of interest in the Scala community about a data structure known as Free or the Free Monad. It allows Scala developers to define internal mini languages, and hence programs, inside their own applications that they can interpret and execute independently as needed. Free behaves much like Stream and Optional do in Java 8. While the map and flatMap methods on Stream capture the transformation steps to be applied to a Stream of data, the map and flatMap methods on Free can capture the sequencing of commands to be executed in a custom program.

This technique, of defining DSLs can be used to enforce SOLID design principles (in particular Dependency Inversion) and allows us to ensure that we don’t mix abstraction levels at critical junctures in our code. While our DSLs lack the syntax for lower or higher levels of abstraction, they force us to keep to the right path. We can also attach different interpreters to our DSLs at runtime, so for example in our unit tests the interpreter can mock the live service calls we’d actually make in a production environment.

While Free itself is an advanced concept, defining DSLs in terms in of Free can be done entirely in terms of pure Java 8 and is arguably a simpler task than defining and combining multiple grammars and parsers using tooling such as Antlr. However I suspect most Java developers will find this article useful as a primer on a concept that may seem alien and even unintelligible when expressed in other languages.

The examples in Part I will be based on a simplified implementation of Free provided in cyclops-react called Unrestricted.

This series of articles / tutorials is based on the Haskell Blog Post by Gabriel Gonzalez called Why Free Monads Matter, and its Java Conversion by Kenji Yoshida.

The article continues below, where we will

define a simple DSL

a program written in that DSL

2 Interpreters for our new DSL that handle program execution differently

Expressing Free in Java

Let’s start by defining a simple program

In this program we want to

output A to the console

ring a bell

output B to the console

terminate

Using the Unrestricted type in cyclops-react (a simplified implementation of Free) we can express and capture this program in pure Java as follows :-

The meat and bones of the program are on the right, and boiler plate to capture this is defined on the left (don’t worry we will explain what is going on here soon).

The variable ‘program’ captures the sequence of commands to be executed. Once the above code is run, we end up with a linked Data Structure that could be visualized something like this;

The returned Unrestricted (free monad) Instance has an embedded reference to the first command in our program which in turn can generate a reference to the next Unrestricted instance and so on. To execute our program all we need do is walk this data structure (using the resume method on Unrestricted) and attach an interpreter to handle each of the commands.

Building an Interpreter

In our interpreter we want to walk the captured data structure and interpret each of the commands. At this stage we know we will have to handle output, bell and done commands. We will be able to put more meat on the bones of our interpreter after introducing a few more concepts.

Free works with Functors

Java 8 introduced a number of Functors (or data types that can transformed via a function) including Stream, Optional and CompletableFuture.

When we make use of the map method on a Stream, we are treating it as a Functor.

We can also define our own Functors, for example to make List behave as a Functor we could define a map method for Lists.

A more general Functor interface could be defined something like this

In cyclops-react the name Functor is reserved for the more abstract Functor type class that leverages psuedo Higher Kinded Types (of which more in part II of this series). The type Transformable is used to define a simpler / easier to use Functor interface -> Link To Transformable class

Commands are Transformable

We can define each of our programs’ commands by defining an appropriate map method. Our first step is to create an appropriate base class for our DSLs’ commands.

Our map method will be used by our Free implementation (Unrestricted) to realize the chain of Unrestricted instances that represent our program. The provided Function will often return an instance of Unrestricted that can lead us to the next Command in the chain (and so on).

Done

This makes defining Done really simple, as Done represents the last stage of the program we can pretty much ignore the supplied Function.

Bell

Bell is a little more complex.

We need to define a generic variable that can represent a path to the next stage (although it is not shown in this example we can also use next to pass data between commands). In practice, the next stage of the program will stored inside Bell as an instance of Unrestricted / Free.

In the map method we need to apply the provided function to our next variable.

Output

The implementation of Output is pretty much identical to that of Bell, except that we also need to capture & store the character that we wish to output to the console.

Commands want to be Free

In our program all our commands are embedded inside instances of Free / Unrestricted. We use a static method to create each command.

To embed an Output instance inside an instance of Free we can create a static method to do that and make use of the ‘LiftF’ method on Unrestricted.

LiftF creates an instance of Unrestricted with an embedded functor (an instance of Transformable). As all our Commands implement Transformable we can pass them to LiftF.

Interpreting our Java Program

Hopefully the boilerplate makes a little more sense now. On the right we are creating instances of our commands embedded inside Free / Unrestricted.

On the left we are defining our program in terms of a Free instance. On the right each command is provided as a parameter to the forEach method — either directly or as the return value from a function.

Ultimately the variable program represents the sequencing of these commands in a data-structure something like a linked chain of Free / Unrestricted instances. Any combination of the commands we have defined can be put together in this way to define any acceptable program. We just need to finish our interpreter to be able to execute it.

Interpreting the Output Command

When we have an output command we want to do 2 things.

We want to append the character defined as an operand to this Command to our output String

We want to interpret the next step of the program

With Java 8 lambdas we can implement the visitor pattern (aka pattern matching in functional languages) for this type really simply. In our visitor we should provide both the captured character and the next step of the program to a user defined function.

We can then make use of this visit method when we handle output by passing it a BiFunction that appends the appropriate character to an output String and recursively interprets the next stage of the program.

Interpreting the Bell Command

We can do the same for our Bell command.

In this case things are even simpler as we only need to capture the next stage of the program.

Our visit method should take a function that accepts the next stage of the program.

public class Bell<T> extends Command<T> {

private final T next;

private Bell(final T next) {

this.next = next;

}

public <R> R visit(final Function<? super T, ? extends R> nextStageFn) {

return nextStageFn.apply(next);

}

//..more code//

}

To interpret each Bell command we can use our visitor to append the word ‘bell’ to our output (or if you are feeling adventurous use some Java library to play a sample of a bell ringing) before making a recursive call to interpret the next stage of the program.

public class StringBuildingInterpreter {

public static <R> String handleBell(Bell<Unrestricted<Command<R>> bell){

//here we should handle the Bell command and because the bell command

//will contain the link

//to the next stage in the program (note the generic type signature)

//we should recursively call interpret

return bell.visit(next -> "bell " + "\n" + iterpret(next));

}

public static <R> String interpret(Unrestricted<R> program){

//walk the Free data structure and handle each command,

//by delegating to the appropriate method

}

//..more code//

}

Implementing the interpret method

The last piece of the puzzle is to implement the interpret method itself. We can use the resume method on Free to interpret the next stage in the program. The returned object also implements a visitor which allows us to pattern match on the result. Resume will either return the current command to be interpreted or a raw result, we can implement matchCommand to handle each command type.

public class StringBuildingInterpreter {

public static <R> String interpret(Unrestricted<R> program){

//walk the Free data structure and handle each command,

//by delegating to the appropriate method

return program.resume(Command.decoder())

.visit(

r -> matchCommand(r),

\_\_->"\n"

);

}

private static <R> String matchCommand(Command<R> command){

if(command instanceof Output)

return handleOutput((Output<Unrestricted<R>>)command);

else if(command instanceof Bell)

return handleBell((Bell<Unrestricted<R>>)command);

else if(command instanceof Done)

return handleDone((Done<R>)command);

throw new IllegalArgumentException("Unknown command " + command);

}

//..more code//

}

For simplicity’s sake, and to avoid introducing another potentially new data type, we can implement matchCommand in terms of a traditional if / else block — but there are cleaner and more functional ways to implement this without risking throwing an exception (which we will cover in part II of this series).

If we run our program via

System.out.println(interpret(program));

We should see the following printed to the console -

emitted A

bell

emitted B

done

Converting from Transformable to Command

One of the downsides of not having Higher Kinded Types means we lose the fact that the functor type is actually a Command (rather than just something that implements Transformable). The purpose of the ‘decoder’ function passed to resume() is to provide a way to convert back from Transformables to Commands. Here we lack full type safety and will make use of a cast.

public final static <T> Function<Transformable<Unrestricted<T>>,Command<Unrestricted<T>>> decoder() {

return c->(Command<Unrestricted<T>>)c;

}

A note on syntax

Unrestricted.comprehensions.forEach is an API that provides ‘for comprehensions’ for Unrestricted free monad instances.

The following code snippets are all equivalent in effect, but using the forEach facility provided by Unrestricted.comprehensions produces much cleaner code.

Unrestricted<Command<Void>> program = output('A').flatMap(a -> {

return bell().flatMap(b ->{

return output(‘B’).map(c->{

return done();

);}

);}

);})))

import static cyclops.control.Unrestricted.comprehensions.forEach;

Unrestricted<Void> program = forEach( output('A'),

\_\_ -> bell(),

\_\_ -> output('B'),

\_\_ -> done());

Get started

ModernNerd Code

DSLs with the Free Monad in Java 8 : Part I

John McClean

John McClean

May 30, 2017 · 8 min read

A program in Free

Recently, there has been an explosion of interest in the Scala community about a data structure known as Free or the Free Monad. It allows Scala developers to define internal mini languages, and hence programs, inside their own applications that they can interpret and execute independently as needed. Free behaves much like Stream and Optional do in Java 8. While the map and flatMap methods on Stream capture the transformation steps to be applied to a Stream of data, the map and flatMap methods on Free can capture the sequencing of commands to be executed in a custom program.

This technique, of defining DSLs can be used to enforce SOLID design principles (in particular Dependency Inversion) and allows us to ensure that we don’t mix abstraction levels at critical junctures in our code. While our DSLs lack the syntax for lower or higher levels of abstraction, they force us to keep to the right path. We can also attach different interpreters to our DSLs at runtime, so for example in our unit tests the interpreter can mock the live service calls we’d actually make in a production environment.

While Free itself is an advanced concept, defining DSLs in terms in of Free can be done entirely in terms of pure Java 8 and is arguably a simpler task than defining and combining multiple grammars and parsers using tooling such as Antlr. However I suspect most Java developers will find this article useful as a primer on a concept that may seem alien and even unintelligible when expressed in other languages.

The examples in Part I will be based on a simplified implementation of Free provided in cyclops-react called Unrestricted.

This series of articles / tutorials is based on the Haskell Blog Post by Gabriel Gonzalez called Why Free Monads Matter, and its Java Conversion by Kenji Yoshida.

The article continues below, where we will

define a simple DSL

a program written in that DSL

2 Interpreters for our new DSL that handle program execution differently

Expressing Free in Java

Let’s start by defining a simple program

In this program we want to

output A to the console

ring a bell

output B to the console

terminate

Using the Unrestricted type in cyclops-react (a simplified implementation of Free) we can express and capture this program in pure Java as follows :-

The meat and bones of the program are on the right, and boiler plate to capture this is defined on the left (don’t worry we will explain what is going on here soon).

The variable ‘program’ captures the sequence of commands to be executed. Once the above code is run, we end up with a linked Data Structure that could be visualized something like this;

The returned Unrestricted (free monad) Instance has an embedded reference to the first command in our program which in turn can generate a reference to the next Unrestricted instance and so on. To execute our program all we need do is walk this data structure (using the resume method on Unrestricted) and attach an interpreter to handle each of the commands.

Building an Interpreter

In our interpreter we want to walk the captured data structure and interpret each of the commands. At this stage we know we will have to handle output, bell and done commands. We will be able to put more meat on the bones of our interpreter after introducing a few more concepts.

Free works with Functors

Java 8 introduced a number of Functors (or data types that can transformed via a function) including Stream, Optional and CompletableFuture.

When we make use of the map method on a Stream, we are treating it as a Functor.

We can also define our own Functors, for example to make List behave as a Functor we could define a map method for Lists.

A more general Functor interface could be defined something like this

In cyclops-react the name Functor is reserved for the more abstract Functor type class that leverages psuedo Higher Kinded Types (of which more in part II of this series). The type Transformable is used to define a simpler / easier to use Functor interface -> Link To Transformable class

Commands are Transformable

We can define each of our programs’ commands by defining an appropriate map method. Our first step is to create an appropriate base class for our DSLs’ commands.

Our map method will be used by our Free implementation (Unrestricted) to realize the chain of Unrestricted instances that represent our program. The provided Function will often return an instance of Unrestricted that can lead us to the next Command in the chain (and so on).

Done

This makes defining Done really simple, as Done represents the last stage of the program we can pretty much ignore the supplied Function.

Bell

Bell is a little more complex.

We need to define a generic variable that can represent a path to the next stage (although it is not shown in this example we can also use next to pass data between commands). In practice, the next stage of the program will stored inside Bell as an instance of Unrestricted / Free.

In the map method we need to apply the provided function to our next variable.

Output

The implementation of Output is pretty much identical to that of Bell, except that we also need to capture & store the character that we wish to output to the console.

Commands want to be Free

In our program all our commands are embedded inside instances of Free / Unrestricted. We use a static method to create each command.

To embed an Output instance inside an instance of Free we can create a static method to do that and make use of the ‘LiftF’ method on Unrestricted.

LiftF creates an instance of Unrestricted with an embedded functor (an instance of Transformable). As all our Commands implement Transformable we can pass them to LiftF.

Interpreting our Java Program

Hopefully the boilerplate makes a little more sense now. On the right we are creating instances of our commands embedded inside Free / Unrestricted.

On the left we are defining our program in terms of a Free instance. On the right each command is provided as a parameter to the forEach method — either directly or as the return value from a function.

Ultimately the variable program represents the sequencing of these commands in a data-structure something like a linked chain of Free / Unrestricted instances. Any combination of the commands we have defined can be put together in this way to define any acceptable program. We just need to finish our interpreter to be able to execute it.

Interpreting the Output Command

When we have an output command we want to do 2 things.

We want to append the character defined as an operand to this Command to our output String

We want to interpret the next step of the program

With Java 8 lambdas we can implement the visitor pattern (aka pattern matching in functional languages) for this type really simply. In our visitor we should provide both the captured character and the next step of the program to a user defined function.

We can then make use of this visit method when we handle output by passing it a BiFunction that appends the appropriate character to an output String and recursively interprets the next stage of the program.

Interpreting the Bell Command

We can do the same for our Bell command.

In this case things are even simpler as we only need to capture the next stage of the program.

Our visit method should take a function that accepts the next stage of the program.

To interpret each Bell command we can use our visitor to append the word ‘bell’ to our output (or if you are feeling adventurous use some Java library to play a sample of a bell ringing) before making a recursive call to interpret the next stage of the program.

Implementing the interpret method

The last piece of the puzzle is to implement the interpret method itself. We can use the resume method on Free to interpret the next stage in the program. The returned object also implements a visitor which allows us to pattern match on the result. Resume will either return the current command to be interpreted or a raw result, we can implement matchCommand to handle each command type.

For simplicity’s sake, and to avoid introducing another potentially new data type, we can implement matchCommand in terms of a traditional if / else block — but there are cleaner and more functional ways to implement this without risking throwing an exception (which we will cover in part II of this series).

If we run our program via

We should see the following printed to the console -

emitted A

bell

emitted B

done

Converting from Transformable to Command

One of the downsides of not having Higher Kinded Types means we lose the fact that the functor type is actually a Command (rather than just something that implements Transformable). The purpose of the ‘decoder’ function passed to resume() is to provide a way to convert back from Transformables to Commands. Here we lack full type safety and will make use of a cast.

A note on syntax

Unrestricted.comprehensions.forEach is an API that provides ‘for comprehensions’ for Unrestricted free monad instances.

The following code snippets are all equivalent in effect, but using the forEach facility provided by Unrestricted.comprehensions produces much cleaner code.

Also note, that Free / Unrestricted programs can be arbitrarily large.

Multiple interpreters can be used

Once you have created one interpreter for your knew DSL, it’s pretty straightforward to add more. The StringBuildingInterpreter we’ve created looks like it could be useful for testing programs written in our new mini-language, but at runtime we might want each command to actually Output to the Console or play a bell.

We can create a new EffectfulInterpreter, based on the old one, that does just that.

import static com.aol.cyclops2.types.mixins.Printable.println;

public class EffectfulInterpreter {

public static <R> void interpret(Unrestricted<R> program){

//walk the Free data structure and handle each command,

//by delegating to the appropriate method

program.resume(Command.decoder())

.visit(

r -> { matchCommand(r); return null;},

\_\_->"\n"

);

}

private static <R> void matchCommand(Command<R> command){

if(command instanceof Output) {

handleOutput((Output<Unrestricted<R>>) command);

return;

}

else if(command instanceof Bell) {

handleBell((Bell<Unrestricted<R>>) command);

return;

}

else if(command instanceof Done) {

handleDone((Done<R>) command);

return;

}

throw new IllegalArgumentException("Unknown command " + command);

}

static <R> void handleOutput(Output<Unrestricted<R>> output){

output.visit((a, next) -> {

println("emitted " + a);

interpret(next);

return null;

});

}

static <R> void handleBell(Bell<Unrestricted<R>> bell){

bell.visit(next ->{

java.awt.Toolkit.getDefaultToolkit().beep();

interpret(next);

return null;

});

}

static <T> void handleDone(Done<T> done){

println("done");

}

}

At runtime we can choose which interpreter should be used, or even leverage both.

public static void main(String[] args){

Unrestricted<Void> program = forEach( output('A'),

\_\_ -> bell(),

\_\_ -> output('B'),

\_\_ -> done());

System.out.println("\*\*\*\*\*\*\*\*\*");

System.out.println("Executing program with StringBuildingInterpreter..");

System.out.println();

System.out.println(StringBuildingInterpreter.interpret(program));

System.out.println("\*\*\*\*\*\*\*\*\*");

System.out.println("Executing program with EffectfulInterpreter..");

System.out.println();

EffectfulInterpreter.interpret(program);

}

DSLs with the Free Monad in Java 8 : Part II

This article builds on DSLs with the Free Monad in Java 8 : Part I and Simulating Higher Kinded Types in Java . In DSLs with the Free Monad we covered the creation of a simple DSL and program using a simpler / Java friendly version of the Free Monad from cyclops-react called Unrestricted.

In this article we will re-implement the same DSL, in a full blown, Higher Kinded version of Free (also from cyclops-react), and still in Java 8. The focus of this article is introducing the building blocks neccessary to implement (or simulate) Higher Kinded Types in Java, we will rework the same examples from part I so that they are encoded at a level of abstraction normally only in seen in languages with more advanced type systems such Scala or Haskell but in plain ol’ Java 8!

Higher Kinded Functors and Commands

In Part I we made use of a Transformable class that represents a type with a map operation (a functor!) and in Simulating Higher Kinded Types in Java we created a Higher Kinded equivalent of this interface.

public interface Transformable<T> {

<R> Transformable<R> map(Function<? super T, ? extends R> fn);

}

//Higher Kinded equivalent

public interface Functor<W> {

<T,R> Higher<W,R> map(Function<? super T, ? extends R> fn, Higher<W, T> ds);

}

By providing a Higher Kinded encoding for our Commands, we can migrate our DSL to use our new Functor interface rather than Transformable. The easiest way to do this (as a first step refactoring), is simply to have Command implement Higher. Our Functor instance can delegate the map operation to the map method on each Command.

//Transformable Command

abstract class Command<T> implements Transformable<T>{

public abstract <R> Command<R> map(Function<? super T, ? extends R> f);

}

//After refactoring to use Higher Kinded Types

abstract class Command<T> implements Higher<µ,T>{

public static class µ {}

public abstract <R> Command<R> map(Function<? super T, ? extends R> f);

public static <T> Command<T> narrowK(Higher<µ,T> higher){

return (Command<T>)higher;

}

}

Defining a Functor instance for our Command type is straight forward as we can simply delegate to our pre-existing map method.

public class CommandFunctor implements Functor<Command.µ> {

@Override

public <T, R> Higher<Command.µ, R> map(Function<? super T,? extends R> f, Higher<Command.µ, T> command) {

return Command.narrowK(fa).map(fn);

}

}

Expressing a Higher Kinded Free in Java

From part I our simple program was

output('A');

bell();

output('B');

done();

output A to the console

ring a bell

output B to the console

terminate

With our new Functor we can define this program using the Higher Kinded Free type in cyclops-react. One last refactoring before we do so is to modify our ‘lifting’ methods to use Free. The required change is pretty small, switch from Unrestricted to Free (passing in the witness type Command.µ) and when we call liftF on Free we also pass in our Command Functor.

public static Unrestricted<Command<Void>> output(final char out){

return Unrestricted.liftF(new Output<>(out, null));

}

//output method refactored to use the Higher Kinded Free

public static Free<Command.µ,Void> output(final char out){

return Free.liftF(new Output<>(out, null),new CommandFunctor());

}

Once we have done this, we are ready to express our program in Free.

import static cyclops.control.Unrestricted.comprehensions.forEach;

Unrestricted<Void> program = forEach( output('A'),

\_\_ -> bell(),

\_\_ -> output('B'),

\_\_ -> done());

//refactoring our program definition to use Free

import static cyclops.typeclasses.free.comprehensions.forEach;

Free<Command.µ,Void> program = forEach( output('A'),

\_\_ -> bell(),

\_\_ -> output('B'),

\_\_ -> done());

Cleaning up the Interpreter

The last piece of the puzzle is to refactor the interpet method to use Free. While we are it we can also take a little time to clean up the code.

//ORIGINAL CODE :: making changes to this code can introduce bugs

private static <R> void matchCommand(Command<R> command){

if(command instanceof Output) {

handleOutput((Output<Unrestricted<R>>) command);

return;

}

else if(command instanceof Bell) {

handleBell((Bell<Unrestricted<R>>) command);

return;

}

else if(command instanceof Done) {

handleDone((Done<R>) command);

return;

}

throw new IllegalArgumentException("Unknown command " + command);

}

Our Interpreters from Part I used the visit method on the type returned from the resume method to ‘pattern match’ and extract the next command from Free. Inside the matchCommand method itself we used a gnarly traditional if/then/else block to determine how to handle the command. There are three different commands so we do an if / then / else check for each one in turn. Not only is this code difficult to read, it also opens up the possibility of bugs and mistakes. The compiler will not help us if we miss an else if statement!

Either types and Pattern matching

cyclops-react provides the visit method as a way to deconstruct (or fold over) the internal state of it’s data types. This is pretty much the same approach as is taken in pattern matching in functional languages (and dual paradigm languages like Scala today and Java in the future). While we wait on the Java compiler team to implement this support for us, we can move forward today by selecting appropriate representative data types for our data.

When data is potentially nullable we can make use of an Optional (or Maybe) type, we can ‘visit’ Optional and execute one function when data is present, or a supplier when it is empty.

If we have a choice of two data types (let’s call them type A and type B) we can make use of an Either (or Xor) type. That is, we can have one thing or the other (Either A or B : Either<A,B>). When we fold / or visit this type, we will provide two functions. One will be executed when type A is present and the other when type B is present.

If we have choice of three types (for example an Output, Bell or Done command) we can define this via an Either3 (Either3<Output,Bell,Done>). Making use of an Either3 inside our Interpreter will significantly improve our code. It will look cleaner and be more robust to future changes. It will be more difficult for an engineer to refactor a change to our logic that drops one of the key conditions.

Let’s add support for our own visitor to the Command interface

abstract class Command<T> implements Higher<µ,T>{

...

public abstract <R> Command<R> map(Function<? super T, ? extends R> f);

//pattern match on the type of Command

public abstract Either3<Output<T>,Bell<T>,Done<A>> patternMatch();

}

Implementing visit is really simple. In each sub-type we return an appropriate instance of Either3. The compiler will help us, we won’t be able to put the wrong type in the wrong the place!

public class Bell<T> extends Command<T> {

private final T next;

private Bell(final T next) {

this.next = next;

}

public Either3<Output<T>,Bell<T>,Done<A>> patternMatch(){

//attempting to slot a Bell instance in left1 or right results in a compile error!

return Either3.left2(this);

}

//..more code//

}

In our interpreter we can replace our unruly if / then / else chain with a type safe and exhaustive alternative call to visit on Command

//REFACTORED CODE :: now the compiler will help ensure correctness!

private static <R> void matchCommand(Command<Free<Command.µ, R>> command){

return command.patternMatch()

.visit(EffectfulInterpreter::handleOutput,

EffectfulInterpreter::handleBell,

EffectfulInterpreter::handleDone);

//no exception to be thrown at the end

}

Part III: Taking this further

Only a few small changes were required to migrate our original example to use the Higher Kinded version of Free. As usual the entire code is available for you to checkout and play with on github here (https://github.com/johnmcclean-aol/unrestricted-example1).

Do join us for the next installment where we will show how pass data between commands in your programs, and also how to interleave the execution of two programs via the zip operator.