Functional Programming (in #scala) for the rest of us

Posted Tue, 30/08/2011 - 18:15 by **ittayd**

fp, fp scala,

Whoever tries to learn Scala immediately encounters talk about Functional Programming.

The first step is having functions as first-class citizens. This opens up new ways of programming, and the Scala collection library if far superior to Java's because of it. There are many examples of how boilerplate loops of creating and populating new collections can be replaced by a succint one liner of using `map`.

In this post I'll try to explain the more advanced concepts of Functors, Applicatives and Monads. I'm doing it starting from everyday OO and building the API and the implementation as we go along.

(Note: This paper was not easy to write and may still be confusing despite my best efforts. I would love to hear your thoughts and suggestions in the comments, or through twitter: @ittayd)

So what are these? In general, they are abstractions or design patterns that allow to work with values that are wrapped in some context.

What is a context? It is just a trait/class that gives access to a value, but adds more meaning. In this post I'll work with the Future context, and I'll show some examples of more towards the end.

So what is the Future context? It is a generic class that is a reference to a value that will be calculated in the future.

We might try an initial design like this:

```
1 trait Future[A] {
2   def get: A
3 }
```

The *get* method blocks until the value is ready. We might use it like so:

```
1 def findUser(name: String): Future[User] = ...
2
3 ....
4
5 val future = findUser(userName)
6 val user = future.get
7 println(user)
```

So findUser does not immediately return a User value, but a Future instance.

Obviously, the above code is not very good. We try to be lazy in findUser by returning a Future, but using get blocks the thread.

We may try to overload get to accept a timeout, or provide a tryGet that throws an exception if the value is not calculated yet. But these lead to both complex implementation of Future as well as complex client code of retrying, which is also sub optimal since it works in a busy loop.

Our second attempt can be to provide a callback that is called once the value is ready:

```
1 trait Future[A] {
2   def onReady(f: A => Unit)
3 }
```

And use it like so:

```
1 val future = findUser(userName)
2 future.onReady{user => prinln(user)}
3 // or future.onReady(println)
```

But on further inspection, this has several drawbacks:

- What if we want to register more than one callback (imagine the future instance being passed around several methods)
- What if we don't just want to print the value and forget about it, but calculate something

based on it, e.g., get the User's age?

About the second point, without knowing any better, we use 'user.get.age', thus blocking the thread. But if consider this a bit we realize that all we need is that given a Future[User] to return a Future[Int]. This keeps the value in the context and thus avoids using 'get'

Functor

If we know FP concepts, we immediately realize that mapping Future[User] to Future[Int] means that Future needs to be a Functor.

First, here's the new design, then the explanation:

```
trait Future[A] {
  def map[B](f: A => B): Future[B]
}
```

The method map accepts a function that takes a value of type A and returns a value of type B. map returns a new instance of Future that can provide this value.
A few things to note:

- f is a pure function: that is, it knows nothing about Future. This means it is reusable in other contexts.
- map doesn't need to apply f immediately (in fact it must not), instead it can return an instance of Future that will apply f only in the last possible moment.

Here's a trivial implementation

```
trait Future[A] {
  // blocks until a value is ready
  def get: A

  // checks if a value is ready
  def isDone: Boolean

  def map[B](f: A => B) = new Future[B] {
    def get = f(Future.this.get)

    def isDone = Future.this.isDone
  }
}

object Future {
  class Concrete[A] extends Future[A] {
    def put(a: A) = // used by whoever created the Future (e.g., findUser) to put a value once calculated
    def get = // concrete implementation
    def isDone = // concrete implementation
  }
  def create[A] = new Concrete[A]
}
```

And used like:

```
def findUser(name: String): Future[User] = {
  val future = Future.create[User]
  // pass future as callback to some method that will use `put` method when the value is ready
  future
}

val user = findUser(userName)
val age: Future[String] = user.map{user => user.age]
// or user.map{_.age}
```

Now that we have a Future holding the user's age, we can wait until the last minute before calling get. E.g., start rendering a result web page, or if we read several users, maybe parallelism kicks into play, etc.

Something important to note here: I'm not concerning myself with avoiding mutation (Future#put) which is something unholy in "pure FP". I'm just taking the concepts that help in my normal, mutateble, OO programming

To summarise: Knowing about the "design pattern" of creating a map method, I could create better Future class.

UPDATE: Here's a description of Functor as a design pattern:
Problem: You have a generic context trait/class C[A]. You wish to allow working with the values

without ruining the abstraction (without using #get or #isDone)
Solution: provide a method map[B](f: A => B): C[B] that creates an instance C[B] that (lazily) implements the interface of C by applying f on the value.

More Contexts

Here are examples of more contexts with explanation of what they mean (the explanation focused on newbies):

- *Option* - This context holds a value that may not exist. This is similar to the way methods return null in Java when there's nothing to return, but clearly documents, using the type, this fact. That is, a method findUser in Java might return null when no such user exists, but then its return type is User, and one needs to read the documentation to find if a return value of null is possible. Instead, the method can return Option[User] which means it can either return a subclass Some containing the user object, or a singleton instance None

- *Either* - This context is similar to Option, but instead of instances being either Some(value) or None, they can be either Right(value) or Left(error), where the error type can be anything. In particular, it can be an Exception. Then a method may return either Right(user) if the user is found or Left(new UserNotFoundException) otherwise
- *List* - This context holds several values and can be thought of the result of a non-deterministic method. That is, a method that has several results.

(if you're interested in seeing implementation of map for these, look in the scala source code for Option#map and List#map, Either is trickier, you need to use either.right.map, or look in the scalaz library)

Generic Programming

Another aspect of using the Functor design pattern (or abstraction) is that this makes client code uniform

What is the gain here, well, look at this client code:

```
val user = findUser(userName)
val age = user.map{user => user.age]
```

Imagine that we change findUser to return Option[User] instead of Future[User] (fortunately, Option has a map method). The client code doesn't change (just needs to be recompiled).

Also, a reader of this code that knows a bit about FP immediately understands that user is in some kind of context and that map will return a new context around the user's age.

Applicative

What happpens if we want to work with several "contextual values". E.g.,:

```
def marry(man: User, woman: User): Family = ....

val joe = findUser("joe")
val jane = findUser("jane")
```

```

```

We want to call a marry on joe and jane. But how can we? We can try to use map:

```
joe.map{joe => jane.map{jane => marry(joe, jane)}}
```

```

```

But since each call to map wraps the result type of the function in a Future, we end up with Future[Future[Family]]

So we need another method to work with functions of higher arity. An initial approach is:

```
def app[B, C](other: Future[B], f: (A, B) => C): Future[C]
```

This works, but what happens if we want to use a function with 3 arguments? 4 arguments?

Knowing the applicative patterns helps here. The key point is that a if we take a function f that accepts n arguments, it can be though of instead as a chain of n functions each accepting one

argument and returning another function, or the final result. In other words, instead of type `f: ($A_1$,$A_2$,...,$A_n$) => R` f can be can instead be viewed `f: $A_1$=>$A_2$=>,...,=>$A_n$=>R`. In fact, Scala has native support for this:

```
scala> def marry(man: User, woman: User): Family = null
marry: (man: User, woman: User)Family

scala> // or, val marry: (User, User) => Family ...

scala> marry _
res0: (User, User) => Family = <function2>

scala> (marry _).curried
res1: (User) => (User) => Family = <function1>
```

Working with 1 argument functions is easier, lets try:

```
scala> joe.map((marry _).curried)
res3: Future[(User) => Family] = // elided
```

What does a Future of a function mean? It means that once the joe future value is set, then marry can be partially applied and we get a function that we can apply on jane.

So now we need a way to apply a function inside a Future to more arguments (jane in this case). So we add a method:

```
trait Future[A] {
  def map...

  def apply[B](f: Future[A => B]): Future[B]
}
```

And now we can use this as:

```
val partial = joe.map((marry _).curried)
val family = jane.apply(partial)

// or:
val family = jane.apply(joe.map((marry _).curried))
```

This solves our problem, but is really ugly. The scheme of passing a function into a method doesn't work very well. Instead, what if we could do this:

```
val futureMarry: Future[User => User => Family] =
Future.create(marry)
val partial: Future[User => Family] = futureMarry.apply(joe)
val family: Future[Family] = partial.apply(jane)

// or:
Future.create(marry).apply(joe).apply(jane)
```

So we start with a curried function and then reduce it by applying it to a single argument each time, until we have the result.

(UPDATE: I've updated the following implementation of apply according to lockster in his blog post. Previously it was a utility trait and implicit conversion)

Looks promising, but for this to work, we need a way for apply to be defined only for Future instances holding a function. Furtunately, Scala has a nice feature called generalized type constraints which allows to constrain a method so it can only be used on instances that correspond to some condition. In our case, the condition is that A will be a function.

```
def apply[B, C](b: Future[B])(implicit ev: A <:< (B => C)) = new
Future[C] {
  def get = Future.this.get(b.get)
  def isDone = Future.this.isDone && b.isDone
}
```

The implicit argument *ev* is an evidence that A is a function (B => C). That is, if calling apply on a future where A is something else, then the compiler will not be able to find a value for ev and the compilation will fail. ev is also an implicit function, so acts as an implicit conversion. So inside apply, Future.this.get(b.get) gets the wrapped value Future.this.get, converts it to a function and applies it to b.get (blocking if the future value is not resolved yet)

Note that that a funtion of the form B => D => E will also be valid since it can be viewed as B => (D => E). That is, the C generic parameter is (D => E).

To lift a function into a future, we create a utility method:

```
object Future {
```

```
  ...

  class Value[A](a: A) extends Future[A] {
    def get = a
    def isDone = true
  }
  def create[A](a: A) = new Value(a)
}

```

Future.create creates a dummy Future that already has the value resolved.

And the usage is:

```
Future.create((marry _).curried).apply(joe).apply(jane)
```

So now we have a way to apply any sort of function of simple values on future values. We do it by putting it in context and then knowing how to tread functions in context.

This code is still not nice. It requied the expression `(marry _).curried`. We can overload Future.create to have instances for functions of different arities, and call curried on them. E.g.:

```
object Future {
  ....

  def create[A, B, C](f: (A, B) => C) = new Value(f.curried)
  def create[A, B, C, D](f: (A, B, C) => D) = new Value(f.curried)
}
```

Or we can create a trait ApplicativeSupport like:

```
trait ApplicativeSupport[AP[_]] {
  def create[A](a: A): AP[A]

  def create[A, B, C](f: (A, B) => C) = create(f.curried)

  def create[A, B, C, D](f: (A, B, C) => D) = create(f.curried)
}

object Future extends ApplicativeSupport[Future] {
  def create[A](a: A) = new Value(a)
}
```

This uses Scala's support for type constructors to provide generic result values of create.

Monads

So far, we've dealt with functions that work on "pure" values, like marry, or User#age. But what happens when we need to deal with functions that return a value in a context. We already used one: findUser, now imagine we have another one:

```
def findProfile(user: User): Future[User]
// or: val findProfile: User => Future[User]
```

Imagine both functions were simple ones:

```
def simpleFindUser(name: String): User
def simpleFindProfile(user: User): Profile
```

Then there's no problem using them:

```
simpleFindProfile(simpleFindUser(userName))
```

But we can't use the result of findUser to findProfile, since the types don't match.

But, similarly to `map`, that accepted f: A=>B, we can define flatMap:

```
trait Future[A] {
  ....
  def flatMap[B](f: A => Future[B]): Future[B]
}
```

And use like so:

```
findUser(userName).flatMap(findProfile)
```

The implementation of flatMap is similar to that of map:

```
trait Future[A] {
  ...
  // note: simplified for beravity. there's a bug here (see below)
  def map[B](f: A => Future[B]) = new Future[B] {
    def get = f(Future.this.get).get

    def isDone = Future.this.isDone && f(Future.this.get).isDone
  }
}
```

The above implementation is short, but has a bug since f may be called multiple times. If it accesses the database, it will access it multiple times. There are two solutions here:

1. add a variable to hold the result of f once used
2. use only pure functions. pure functions do not create side effects (such as accessing a database) and so can be called once or multiple times with the same arguments, since they will always return the same results. Even if their calculation is heavy, their first invocation can be cached without worry that the cache will be stale later (e.g., due to changes in the database).

Approach 1 is left as an excersize to the reader. Approach 2 requires the use of the IO monad, and I hope I gave enough incentive to go find more about it.

And back to applicatives: recall that our problem with Functors was that map can't support the use of functions whose arity is larger than 1. But every monad is also an applicative (that is, we can define apply in terms of flatMap. An excercise left for the reader). So we can now do:

```
joe.flatMap{joe => jane.map{jane => marry(joe, jane)}}
```

This is a bit messy, which is why Scala has for comprehensions:

```
for (man <- joe; woman <- jane) yield marry(man, woman)
```

This translates to exactly the previous expression.

So scala's for comprehensions are really not about loops, but about monadic function application! And by implementing map and flatMap we gain Scala's support to make client code even more consistent, readable and reusable.

Summary

The use of Functors, Applicative and Monad (as well as other classes of types) is just like following the GoF design patterns. It helps us creates better code for dealing with values in a context and in a universal way. Code that uses map/apply/flatMap need not change when the context changes, the context type do not prolifirate unneedly and its meaning is obvious to those familiar with the concepts.

A Word About Implementation

(well, maybe more than a word..)

Throughout this article, the design was OO based. I've added methods to Future and when required created an implicit conversion to another class. This is nice for explaining things from an OO perspective, but not very scalable. Once you start learning FP concepts you realise that Future,Applicative and Monad are just the tip of the iceberg. Adding more and more methods will make our API very bloated. Furthermore, methods like map, flatMap, apply are more related to how the future object is used, rather than to its semantics (contrary to get and isDone which are).

Inheritance will give us little gain. The methods are specific to each type (map for Option is not implemented like map for Future) and creating a huge inheritance that is again concerns usage semantics instead of entity semantics (a Future isn't a is-a Functor, rather, it has Functor properties)

And what happen if we reallize a 3rd party implementation is a Functor? How can we add `map` to it? For example, Scala's Either doesn't have a map method.

Finally, in one instance we required implicit conversion, which we can't automatically get through inheritance. Requiring users to import the right implicit conversion before using apply is confusing and is not self-revealing. The user has to know they exists to use them (where to import from). If we try to create an FPSupport object with all possible implicit conversions then it may just make things more confusing.

So inheritance doesn't help us here. Again, following FP concepts, we can achieve all the above using something called type classes. But this is beyond the scope of this article

And of course, this article is not complete without pointing to the "reference" implementation of FP in Scala: scalaz

A Word About Laws

Implementations of Functor, Applicative and Monad must follow some laws to be vallid. In particular, so they don't influence the result. I didn't go over them, to avoid confusion, and since usually the natural implementation follows them, but readers should read about them if they want to create their own implementations

Further Reading

I really enjoyed these blog posts:
The essense of the iterator pattern
Datatype generic programming in Scala

The blogs where these articles are found are great in general for FP as well as Apocalisp

COMMENTS

by kilburn - 31/08/2011 - 00:49
Hi ITTAYD,
Although the article looks very interesting, I must say that I am having a hard time understanding it. First of all, this can totally be my fault (I am not acquainted with neither scala nor functional design patterns), so sorry if I waste your time.
Anyway, I will remark the questions that arised when I was reading the article, because they might help you improve it (if you want) for dumb readers such as myself. Feel free to ignore them though! :)
Our second attempt can be to provide a callback that is called once the value is ready:

```
trait Future[A] {
    def onReady(f: A => Unit)
}
```

And use it like so:

```
val future = findUser(userName)
future.onReady{user => prinln(user)}
// or future.onReady(println)
```

What is this "Unit" type? Quick googling seems to show that it is the equivalent of "void" in java, isn't it?

What if we don't just want to print the value and forget about it, but calculate something based on it, e.g., get the User's age? [...] without knowing any better, we use 'user.get.age', thus blocking the thread. But if consider this a bit we realize that all we need is that given a Future[User] to return a Future[Int]. This keeps the value in the context and thus avoids using 'get'
Since the callback receives a "real" User object (not a Future), why can't the callback just "println(user.age)" or something similar? Maybe what I am missing is a small client code block showing the real benefit of Futures in general though.

```
object Future {
    class Concrete[A] extends Future[A] {
        def put(a: A) = // concrete implementation
```

```
        def get = // concrete implementation
        def isDone = // concrete implementation
    }
    def create[A] = new Concrete[A]
}
```

[..]

Something important to note here: I'm not concerning myself with avoiding mutation (Future#put) which is something unholy in "pure FP". I'm just taking the concepts that help in my normal, mutateble, OO programming

What is the purpose of "put(a: A)" here? Is it doing anything at all in the example? If it is, please explain what or otherwise remove it.

What does a Future of a function mean? It means that once the joe future value is set, then marry can be partially applied and we get a function that we can apply on jane.

So now we need a way to apply a function inside a Future to more arguments (jane in this case). So we add a method:

```
trait Future[A] {
  def map...

  def apply[B](other:&amp;nbsp;Future[): Future[B]
}
```

There is a syntax error in the definition of "apply". I would say that it should read:

```
   def apply[B](other: Future[A]): Future[B]
```

Yet I am not sure if the type I've written is right.

And that's it for now...

by Arg - 31/08/2011 - 04:38

> *Since the callback receives a "real" User object (not a Future), why can't the callback just "println(user.age)" or something similar? Maybe what I am missing is a small client code block showing the real benefit of Futures in general though.*

Hello kilburn,

i hope i won't say anything non-sensical but i think the idea is that method .age on User object returns future aswell. Thus you can't just call user.age. Imagine that getting user age would require another call to database. In that case it would make sense to represent the value as future. Or at least thats how i understood it.

by ittayd - 31/08/2011 - 09:04

user.age returns Int. But like I wrote to kilburn, printing inside the callback, while good for a toy example, is not a solution in a real setting, since you can't control when it will happen and printing can't influence the program logic (just useful for logging and such), in other cases we may need the age value (I gave kilburn an example)

by Arg - 31/08/2011 - 14:38

Sorry, i now realize how little sense my previous comment makes. I shouldn't post at 4a.m. i guess :)

by ittayd - 31/08/2011 - 08:59

A Unit type is the analegous of void in Java, but not equivalent. In FP, every expression must return a value (otherwise, it is just side effecting), so even an expression that has nothing to return must return something. The Unit class has a single instance, named (), which is what the expression returns. So you can write: `def foo = ()` which defined a function that returns Unit

If you try to println(user.age), when will that happen? Somewhere in the future, probably not what you want (you'd have sporadic prints appearing unexpectately). Furthermore, println is "use and forget". What if you want to do something with the age value. E.g., you read a list of users and want to create a table listing their name and age. With "normal" OO, you might create a loop over the list and call #get on each User instance to retrieve the age and name and create a table row. With the functor pattern, you'd manipulate Future[User] to become Future[String] (the string is the <tr> element) and return a List[Future[String]]. Further manipulation using FP concepts allow you to create a Future[List[String]] (while keeping everything lazy). Then, the call to #get can be delayed to the last minute, hopefully by then some of the values are already resolved, so #get will not block.

put(a:A) is used by the code that created the Future instance in the first place to put a value in it. So findUser creates a Future[User], but there's no value. It keeps a handle to the reference and when the database query returns, calls put to put the user instance

I'll correct the post with explanation about put and apply

by Arg - 31/08/2011 - 04:59

Hello ittayd,

Thanks for a great article. However, could you please elaborate on what exactly is a 'functor'? I understood it 'has something to do with wrapper around a value and possibility to apply function to that value'. But is it the Future object or is it the map method? (I hope it's understandable what i ask about...can i say "Fututre is a functor" or should i say "Future has a functor map"?)

Also, a bit of offtopic. Could you please elaborate on what the 'origin=>' means? Or at least point me to a right direction? It's really hard to look up all these scala details when one doesn't know what's their name ^_^

Thanks!

Regards,

Tomas Herman

by ittayd - 31/08/2011 - 09:19

A Functor is a design pattern that suggests you add a method map to classes/traits of the form C[A]. I've updated the article to formulate this in terms of a design pattern (problem-solution). I guess the most accurate way to describe is "Future has the Functor capabilities", which means it has a map method. In strict FP terms, map is the Functor (a functor is a way to map values into other values, in this case String to Future[String], which is done by Future.create, and also maintain functions, so f:A =>B is mapped to Future[A] => Future[B], which is what map does, in an OO setting)

About 'origin =>'. The term is "self type" or "self reference". It is a mechanism in Scala that allows to alias 'this' to something else. This is useful when dealing with inner classes, if you want to access the enclosing class. The alternative is to use Future.this. The feature is much more powerful, since it also allows you to give a type to such references, thus making sure a trait is only composed in a class that also uses another trait. Search for "cake pattern" if you want to see a use case.

by Arg - 31/08/2011 - 14:39

Thanks a lot for the explanation. It's a lot clearer now!

Login Or Register To Post Comments

by nuttycom - 03/09/2011 - 22:46

Isn't the signature of apply supposed to be def apply[B](f: Future[A => B]): Future[B]? The way it's written currently, the only sane implementation given the type signature is simply to return the argument.

Login Or Register To Post Comments

by ittayd - 03/09/2011 - 22:53

Oops... Thanks. Fixed.

Login Or Register To Post Comments

by nuttycom - 04/09/2011 - 08:09

Also, I just read your update in the summary. You should be aware that essentially all Future implementations in Scala these days are monadic - Akka's are, Twitter's are, Scalaz's are, BlueEyes's are...

Login Or Register To Post Comments

by ittayd - 04/09/2011 - 08:46

Of course, that's the whole point, that once you get to know this "design pattern" you find it is very useful. I took Future, since the value is both there and not there, so things like 'get' don't work well (in Option, you might be content with checking isEmpty and using get). I didn't mean to suggest I've discovered something that others didn't.

Login Or Register To Post Comments

by nuttycom - 06/09/2011 - 01:47

Yup. In fact, in some of the Future implementations out there (blueeyes, for example) no blocking 'get' is even provided, and this improves your code because it forces *all* computation to be done asynchronously once you're in a Future context. By taking away the option to block, you end up just taking away something you don't really want to do anyway.

Login Or Register To Post Comments