**Validation for Free in Scala**

Due to the complexity of business data, much effort has been spent on data validation. In Scala, using applicatives to build validation was proposed and is widely considered an effective way. In this article, instead of using applicatives, we borrow ideas from free monads, and build the validations “for free”, and accumulate the validation results. We discuss the approach and show the implementation through encoding.

We will not elaborate on how free monad and applicatives validation are modelled in Scala. To understand Free Monad, please refer to [this](https://typelevel.org/cats/datatypes/freemonad.html) and [this](https://underscore.io/blog/posts/2015/04/14/free-monads-are-simple.html). To understand how validation is modelled using applicative functors, please refer to [this](https://www.innoq.com/en/blog/validate-your-domain-in-scala/) and [this](https://www.fyber.com/2018/06/29/introduction-to-validation-on-scala/). We will only focus on how validation can be modelled as free monad, and the benefits it brings.

**Validation in Scala**

Validation can be in different forms when error(s) are detected. Validation can return immediately when first error (or exception) has been encountered, with or without carrying the error or exception message to the call site with the validation result. This is called fast fail validation, in which the validation does not validate all the business rules and only zero or one error is returned, and the process shall be cut short upon first error. This simple form of validation is sometimes considered insufficient, as a full validation is not carried out with accumulated errors. In reality, it is widely used in application development. Monadic entities in Scala, such as Option, Try, and Either, are very handy to use.

Validating all the business rules, and accumulating errors, is a very different business. Applicative functors are proposed and they have effectively solved this problem. Popular Scala libraries such scalaz and cats all provide applicatives validation support. Readers can refer to scalaz.Validation API and cats.data.Validated API for more details.

However, many Scala projects do not use either of these libraries for various reasons; Scala at this stage does not have native support of applicative for comprehensions, although it has been discussed to include it in Scala 3 (Dotty). Therefore, a native validation approach without leaning on any third party libraries is very appealing. In this article, we introduce an approach inspired by free monad. This approach implicitly lifts the validators into monads (for free), and builds an interpreter to execute the monads, and accumulate the validation errors.

**The Validators and Free Monads**

In our approach, validation is a composition of validator executions; each validator represents validation of each business rule, they are composed by a for-comprehension. This is where Free Monad can come into play - validator can be a monad by being lifted to a monad implicitly “for free”. This allows our validation to enjoy all the benefits that free monad can offer – stack free and natural transformation.

First, we discuss the fast failing validation framework and its limitation.

Suppose we need to validate a person’s name and age before saving the person to the database:

**case class** Person(name: String, age: Int){  
 **def** validateName= **if** (name.isEmpty) None **else** Some(**"Success"**)  
 **def** validateAge = **if** (age < 18) None **else** Some(**"Success"**)  
}  
  
**def** save(person: Person): Boolean = {  
 *println*(**s"save $**{person.name} **at age $**{person.age}**"**)  
 **true**}  
  
**val** *person* = *Person*(**"Michael"**, 20)  
**for** {  
 \_ <- *person*.validateName  
 \_ <- *person*.validateAge  
} **yield** *save*(*person*)

There are two problems in the above encoding.

* Validation returns None when encounters error, it does not carry the error to the caller.
* For comprehension desugars the above code into a chain of flatMap followed by a map:

*person*.validateName  
 .flatMap((\_: String) =>  
 *person*.validateAge  
 .map((\_: String) => forcomprehension.*save*(forcomprehension.*person*))  
 )

When validateName returns None, the process stops short - his is fast failing validation.

Validators conform to a simple trait:

**sealed trait** Validator[A] {  
 **def** validate: Option[Error]  
 **def** unbox: A  
}

Each validator implements its own validation rule:

**case class** NameValidator(name: String) **extends** Validator[String] {  
 **def** validate = **if** (name.isEmpty) *Option*(NameError) **else** None  
 **def** unbox: String = name  
}  
  
**case class** AgeValidator(age: Int) **extends** Validator[Int] {  
 **def** validate = **if** (age >= 18) None **else** Some(AgeError)  
 **def** unbox: Int = age  
}

They are lifted to free monads implicitly:

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

The interpreter executes the validators accordingly:

**val** interpreter = **new** Executor[Validator] {  
 **override def** unbox[A](fa: Validator[A]) = fa.unbox  
 **override def** exec[A](fa: Validator[A]) = fa.validate  
}

**def** validate[F[\_], A](prg: Free[F, A], interpreter: Executor[F]): List[Error] = {  
 **def** go(errorList: List[Option[Error]], prg: Free[F, A]): List[Option[Error]]= prg **match** {  
 **case** *Return*(a) => errorList  
 **case** *FlatMap*(sub, cont) => go(interpreter.exec(sub) :: errorList, cont(interpreter.unbox(sub)))  
 }  
 go(*List*.*empty*[Option[Error]], prg).flatten  
}

The interpreter is the glue between the data, the error, the data validators and the free monads. Notice two details in interpreter:

* The interpreter provides an unbox function, it is used in no-error case. When a None type is returned, unbox can find the data being validated in type safe way, and continue the validation.
* Unlike the interpreter in free monad, in which the continuation of the process is through monadic operations:

executor.exec(sub).flatMap(x => *validateAndRun*(cont(x), executor))

In our validation interpreter, continuation of the process is guaranteed by following the validators.

**The Validation Composition**

Just as the work flow modelling in free monads, the validation flow is modelled through for-comprehension. For instance, we will call save(person) if name validation and age validation all pass, otherwise, we print out the errors:

**val** *validation* = **for** {  
 \_ <- *NameValidator*(*person*.name)  
 \_ <- *AgeValidator*(*person*.age)  
} **yield** ()

*validate*(*validation*, interpreter) **match** {  
 **case** *Nil* => *save*(*person*)  
 **case** errors => errors foreach *println*}

The implementation can be found at [Validation For Free](https://github.com/michaelw123/validation)

Conclusion

Free Monad allows

From [here](https://github.com/michaelw123/free-monad-enhanced), you can also find the basic free monad implementation, an enhanced free monad implementation, and varieties of fast fail validation implementations introduced in this article.