Functional Effects

Part 1

learn about functional effects through the work of

Debasish Ghosh
@debasishg

slides by @philip_schwarz
slideshare https://www.slideshare.net/pjschwarz

Functional thinking and implementing with pure functions is a great engineering discipline for your domain model. But you also need language support that helps build models that are **responsive to failures, scales well with increasing load, and delivers a nice experience to users**. Chapter 1 referred to this characteristic as being reactive, and identified **the following two aspects that you need to address in order to make your model reactive**:

- **Manage failures**, also known as **design for failure**
- **Minimize latency** by **delegating long-running processes to background threads without blocking the main thread of execution**

In this section, you'll see how **Scala offers abstractions that help you address both of these issues. You can manage exceptions and latency as effects that compose along with the other pure abstractions of your domain model. An effect adds capabilities to your computation so that you don't need to use side effects to model them**. The sidebar "What is an effectful computation?" details what I mean.

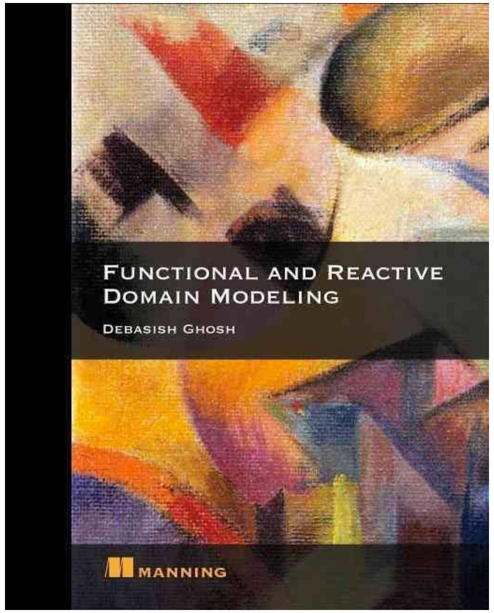
Managing exceptions is a key component of reactive models—you need to ensure that a failing component doesn't bring down the entire application. And **managing latency is another key aspect that you need to take care of**—unbounded latency through **blocking calls** in your application is a severe antipattern of good user experience. Luckily, **Scala covers both of them by providing abstractions** as part of the standard library.

What is an effectful computation?

In functional programming, **an effect adds some capabilities to a computation**. And because we're dealing with a statically typed language, these **capabilities** come in the form of more power from the type system. **An effect is modeled usually in the form of a type constructor that constructs types with these additional capabilities**.

Say you have any type A and you'd like to add the **capability** of aggregation, so that you can treat a collection of A as a separate type. You do this by **constructing a type** List[A] (for which the corresponding type constructor is List), **which adds the effect of aggregation on** A. Similarly, you can have **Option**[A] that **adds the capability of optionality for the type** A.

In the next section, you'll learn how to **use type constructors such as Try and Future to model the effects of exceptions and latency, respectively**. In chapter 4 we'll discuss **more advanced effect handling using applicatives and monads**.



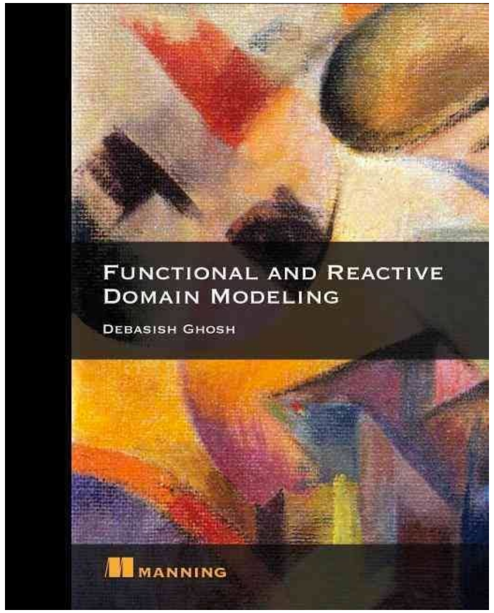
Debasish Ghosh
@debasishg

2.6.1 Managing effects

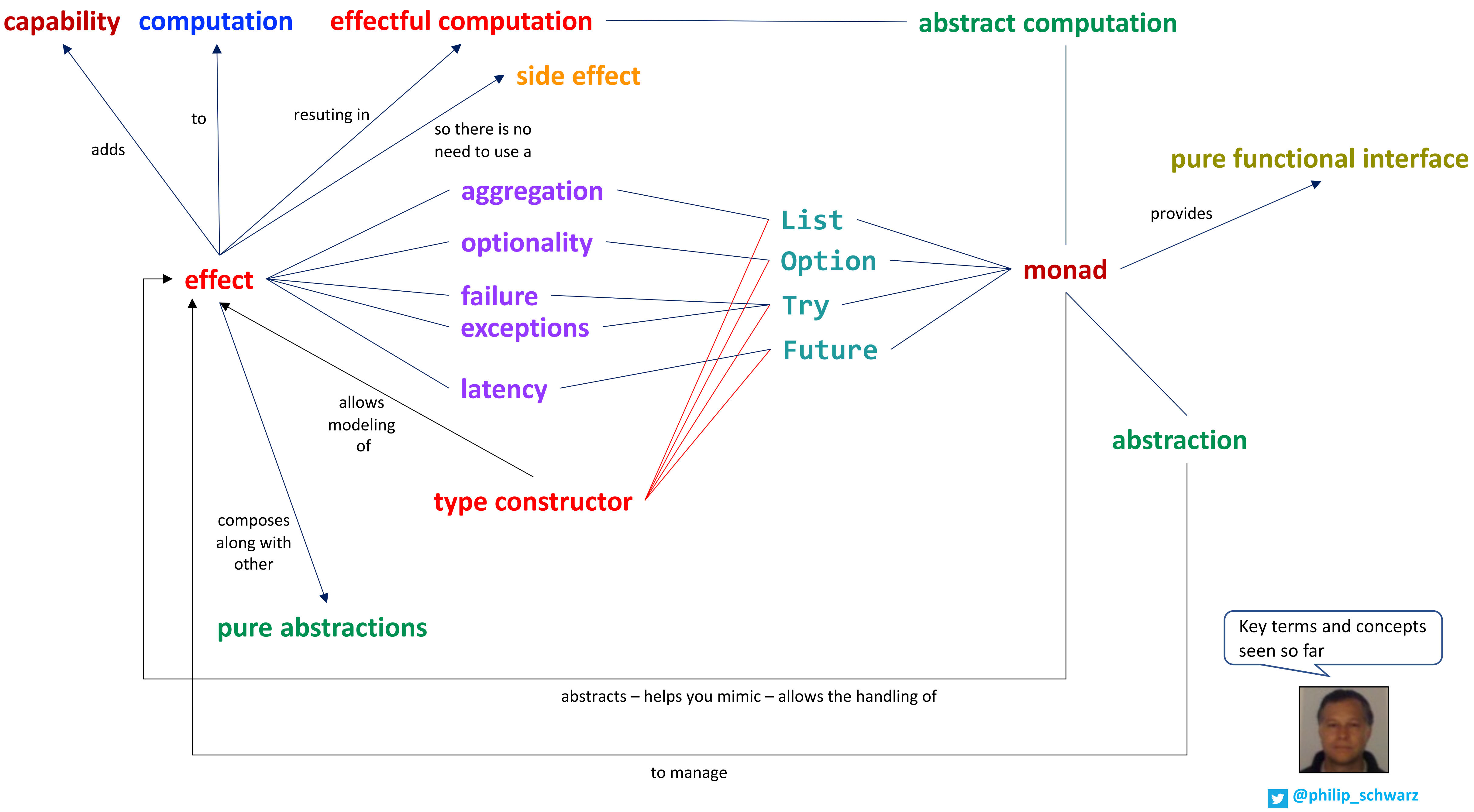
When we're talking about exceptions, latency, and so forth in the context of making your model reactive, you must be wondering **how to adapt these concepts into the realm of functional programming**. Chapter 1 called these side effects and warned you about the cast of gloom that they bring to your pure domain logic. Now that we're talking about managing them to make our models reactive, **how should you treat them as part of your model so that they can be composed in a referentially transparent way along with the other domain elements**?

In Scala, you treat them as effects, in the sense that you abstract them within containers that expose functional interfaces to the world. The most common example of **treating exceptions as effects in Scala is the `Try` abstraction**, which you saw earlier. `Try` provides a sum type, with one of the variants (`Failure`) **abstracting the exception that your computation can raise**. **`Try` wraps the effect of exceptions within itself and provides a purely functional interface to the user**. In the more general sense of the term, **`Try` is a monad**. There are some other examples of **effect handling** as well. **Latency is another example, which you can treat as an effect—instead of exposing the model to the vagaries of unbounded latency, you use constructs such as `Future` that act as abstractions to manage latency**. You'll see some examples shortly.

The concept of a **monad** comes from category theory. This book doesn't go into the theoretical underpinnings of a **monad** as a category. It focuses more on **monads as abstract computations that help you mimic the effects of typically impure actions such as exceptions, I/O, continuations, and so forth while providing a functional interface to your users**. And we'll limit ourselves to **the monadic implementations that some of the Scala abstractions such as `Try` and `Future` offer.** The only takeaway on **what a monad does** in the context of functional and reactive domain modeling **is that it abstracts effects and lets you play with a pure functional interface that composes nicely with the other components**.



Debasish Ghosh
@debasishg



capability
computation
effectful computation
abstract computation
side effect
adds
to
resuting in
so there is no need to use a
pure functional interface
provides
aggregation
optionality
failure
exceptions
latency
effect
List
Option
Try
Future
monad
allows modeling of
type constructor
abstraction
composes along with other
pure abstractions
Key terms and concepts seen so far
abstracts – helps you mimic – allows the handling of
to manage
@philip_schwarz

Managing Failures

Listing 2.13 shows **three functions, each of which can fail** in some circumstances. **We make that fact loud and clear—instead of BigDecimal, these functions return Try[BigDecimal].**

You've seen **Try** before and know how it **abstracts** exceptions within your Scala code. Here, **by returning a Try, the function makes it explicit that it can fail**. If all is good and happy, you get the result in the **Success** branch of the **Try**, and if there's an exception, then you get it from the **Failure** variant.

The most important point to note is that **the exception never escapes from the Try as an unwanted side effect to pollute your compositional code**. Thus you've achieved the first promise of the two-pronged strategy of failure management in Scala: **being explicit about the fact that this function can fail**.

```
def calculateInterest[A <: SavingsAccount](account: A, balance: BigDecimal): Try[BigDecimal] = ???

def getCurrencyBalance[A <: SavingsAccount](account: A): Try[BigDecimal] = ???

def calculateNetAssetValue[A <: SavingsAccount](account: A, ccyBalance: BigDecimal, interest: BigDecimal): Try[BigDecimal] = ???

val account: SavingsAccount = ???

val result: Try[BigDecimal] = for {
  balance ← getCurrencyBalance(account)
  interest ← calculateInterest(account, balance)
  value ← calculateNetAssetValue(account, balance, interest)
} yield value

result match {
  case Success(v)  => ??? // .. success
  case Failure(ex) => ??? // .. failure
}
```

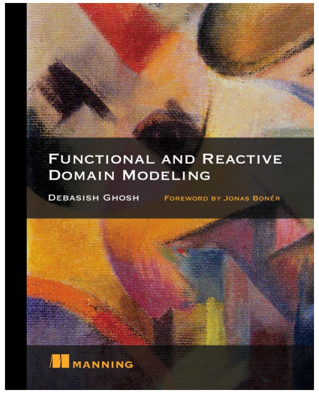
But what about the promise of compositionality? Yes, **Try** also gives you that by being a **monad** and offering a lot of higher-order functions. Here's **the flatMap method of Try that makes it a monad and helps you compose with other functions that may fail**:

```
def flatMap[U](f: T => Try[U]): Try[U]
```

You saw earlier how **flatMap binds together computations and helps you write nice, sequential for-comprehensions** without forgoing the goodness of expression-oriented evaluation. You can get the same goodness with **Try** and **compose code that may fail**:

This code handles all exceptions, composes nicely, and expresses the intent of the code in a clear and succinct manner. This is what you get when you have powerful language **abstractions** to back your domain model. **Try is the abstraction that handles the exceptions, and flatMap is the secret sauce that lets you program through the happy path**. So you can now have domain model elements that can throw exceptions—just **use the Try abstraction for managing the effects and you can make your code resilient to failures**.

Debasish Ghosh
@debasishg



Functional and Reactive Domain Modeling

Managing Latency

Just as Try manages exceptions using effects, another abstraction in the Scala library called Future helps you manage latency as an effect. What does that mean? Reactive programming suggests that our model needs to be resilient to variations in latency, which may occur because of increased load on the system or network delays or many other factors beyond the control of the implementer. To provide an acceptable user experience with respect to response time, our model needs to guarantee some bounds on the latency.

The idea is simple: **Wrap your long-running computations in a Future. The computation will be delegated to a background thread, without blocking the main thread of execution.** As a result, the user experience won't suffer, and you can make the result of the computation available to the user whenever you have it. **Note that this result can also be a failure, in case the computation failed—so Future handles both latency and exceptions as effects.**

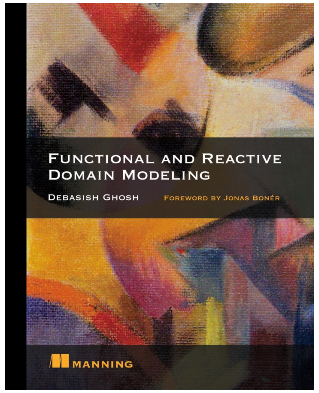
```
def calculateInterest[A <: SavingsAccount](account: A, balance: BigDecimal):Future[BigDecimal] = ???

def getCurrencyBalance[A <: SavingsAccount](account: A): Future[BigDecimal] = ???

def calculateNetAssetValue[A <: SavingsAccount](account: A, ccyBalance: BigDecimal, interest: BigDecimal): Future[BigDecimal] = ???

val account: SavingsAccount = ???
implicit val ec: ExecutionContext = ???

val result: Future[BigDecimal] = for {
  balance ← getCurrencyBalance(account)
  interest ← calculateInterest(account, balance)
  value ← calculateNetAssetValue(account, balance, interest)
} yield value

result onComplete {
  case Success(v)  => ??? //.. success
  case Failure(ex) => ??? //.. failure
}
```

Future is also a monad, just like Try, and has the flatMap method that helps you bind your domain logic to the happy path of computation… imagine that the functions you wrote in listing 2.13 involve network calls, and thus there's always potential for long latency associated with each of them. As I suggested earlier, **let's make this explicit to the user of our API and make each of the functions return Future:**

By using flatMap, you can now compose the functions sequentially to yield another Future. The net effect is that **the entire computation is delegated to a background thread, and the main thread of execution remains free.** Better user experience is guaranteed, and you've implemented what the reactive principles talk about—systems being resilient to variations in network latency. **The following listing demonstrates the sequential composition of futures in Scala.**

Here, result is also a Future, and you can plug in callbacks for the success and failure paths of the completed Future. If the Future completes successfully, you have the net asset value that you can pass on to the client. If it fails, you can get that exception as well and implement custom processing of the exception.

Debasish Ghosh
@debasishg



Functional and Reactive Domain Modeling

```scala
def calculateInterest[A <: SavingsAccount](account: A, balance: BigDecimal): Try[BigDecimal] = ???

def getCurrencyBalance[A <: SavingsAccount](account: A): Try[BigDecimal] = ???

def calculateNetAssetValue[A <: SavingsAccount](account: A, ccyBalance: BigDecimal, interest: BigDecimal): Try[BigDecimal] = ???

val account: SavingsAccount = ???

val result: Try[BigDecimal] = for {
  balance <- getCurrencyBalance(account)
  interest <- calculateInterest(account, balance)
  value <- calculateNetAssetValue(account, balance, interest)
} yield value

result match {
  case Success(v)  => ??? // .. success
  case Failure(ex) => ??? // .. failure
}
```

Managing Failures

```scala
def calculateInterest[A <: SavingsAccount](account: A, balance: BigDecimal): Future[BigDecimal] = ???

def getCurrencyBalance[A <: SavingsAccount](account: A): Future[BigDecimal] = ???

def calculateNetAssetValue[A <: SavingsAccount](account: A, ccyBalance: BigDecimal, interest: BigDecimal): Future[BigDecimal] = ???

val account: SavingsAccount = ???
implicit val ec: ExecutionContext = ???

val result: Future[BigDecimal] = for {
  balance <- getCurrencyBalance(account)
  interest <- calculateInterest(account, balance)
  value <- calculateNetAssetValue(account, balance, interest)
} yield value

result onComplete {
  case Success(v)  => ??? //.. success
  case Failure(ex) => ??? //.. failure
}
```

Managing Latency

That was great. **Functional and Reactive Domain Modeling** is a very nice book.

I'll leave you with a question: are **`Try`** and **`Future`** lawful **monads**? See the following for an introduction to **monad laws** and how to check if they are being obeyed:

Monad Laws Must be Checked slideshare
https://www.slideshare.net/pjschwarz/monad-laws-must-be-checked-107011209

@philip_schwarz