ESSENTIAL EFFECTS

ADAM ROSIEN

Essential Effects

Adam Rosien

Essential Effects

Version f7d80fc @ 2024-03-14.

Published by Inner Product LLC.

More information about this book and its associated course may be found at https://essentialeffects.dev.

Created in Asciidoctor with diagrams generated by Graphviz and PlantUML.

Cover and selected illustrations by @impurepics.

For the helpers:

> *Noel Welsh, my business partner and explainer par excellence,*
>
> *and the kind and generous folk of the Cats Effect community.*

Please provide feedback!

First of all, readers, thank you! Your input is invaluable to improve this book. I'm interested in anything you may want to share, whether you read just one chapter, or the entire book. If you're not sure what to share, here are some ideas:

- "I don't really understand the <*some example*> in Chapter N."
- "Doesn't <*technical term*> actually mean <*something else*>?"
- "Describing <*some concept*> like that really helped me understand it!"

Additionally, would it be ok to share your feedback as a "blurb" or testimonial? If not, that's ok, but if so, it would really help! It could be something as simple as "this book is great!", or "you simply *must* buy your team lots of copies", etc.

Please email any feedback to adam+effects@inner-product.com.

Want to buy the book?

Use discount code `ee-ce3` or visit the book purchase page and get a 33% discount to celebrate the recent update of the book to Cats Effect 3!

Preface

Acknowledgements

I would like to thank the many people who have helped in the creation of this book.

Thank you to Mansur Ashraf, who first suggested building a course about Cats Effect, which then expanded into this book.

Special thanks to the members of the Cats Effect community, especially Gavin Bisesi (Daenyth), Christopher Davenport (ChristopherDavenport), Luka Jacobowitz (LukaJCB), Fabio Labella (SystemFw), Rob Norris (tpolecat), Ryan Peters (sloshy), Michael Pilquist (mpilquist), and Daniel Spiewak (djspiewak). You are a treasure.

Thank you to all the reviewers and proofreaders: Charles Adetiloye, Roman Arkharov, Samir Bajaj, Gavin Bisesi, Guillaume Bogard, Bogdan C., Christopher Davenport, Francisco Diaz, Justin Du Coeur, Saskia Gennrich, Debasish Ghosh, Juan Manuel Gimeno Illa, Daniel Hinojosa, Matt Hughes, Lars Hupel, Marc Ramírez Invernon, Sándor Kelemen, Jakub Kozłowski, Fabio Labella, Chris Lan, Jian Lan, Zachary McCoy, Juan Méndez, Renghen Pajanilingum, Ryan Peters, Philip Schwarz, Eric Swenson, Bartłomiej Szwej, Noel Welsh, Leif Wickland, Yevhenii Zelenskyi, and <*your name here*>.

About this book

Cats Effect[1] is a library that makes it easy to write code that effectively uses multiple cores and doesn't leak resources. This makes building complex applications, such as highly concurrent services, much more productive. This book aims to introduce the core concepts in Cats Effect, giving you the knowledge you need to go further with the library in your own applications.

This book is not, however, a detailed dive into every aspect of Cats Effect. Our aim is to give you the understanding you need so you can rapidly apply it, while setting you up to learn any additional details on your own if needed.

Essential Effects will teach you to:

- Understand the meaning and role of side effects and effects.
- Understand how to encapsulate side effects in a safer form.
- Use `parMapN` and other combinators to run effects in parallel.
- Fork independent work into concurrent tasks, then cancel or join them.

- Learn how to separate CPU-bound work from blocking, I/O-bound work.
- Integrate callback-based code, like `scala.concurrent.Future`, into a safer, effect-based interface.
- Build and combine leak-proof resources for applications.
- Test code that performs multiple effects like concurrency and I/O.

The design of the Cats Effect library uses typeclasses to encode concepts like parallelism, concurrency, and so on. However, rather than programming with an abstract effect type that uses typeclass constraints—a perfectly valid programming technique!—this book uses the concrete `cats.effect.IO` type as the main vehicle to discuss and demonstrate programming with effects.[2]

A functional programming reading list

While there are many excellent books focusing on functional programming in Scala, we specifically recommend the following books as a *functional programming reading list* that will guide you step-by-step, from beginner to expert.

For beginners, or folks new to Scala:

***Creative Scala* by Dave Gurnell and Noel Welsh [1]**

The book for new developers who want to learn Scala and have fun.

***Essential Scala* by Noel Welsh and Dave Gurnell [2]**

Learn to write robust, performant, idiomatic Scala. A focused guide for established developers.

For more advanced concepts:

***Essential Effects* by Adam Rosien**

How to safely create, compose, and execute effectful Scala programs using the Typelevel Cats Effect library.

***Scala with Cats* by Noel Welsh and Dave Gurnell [3]**

Dive deep into functional patterns using Scala and Cats. For experienced Scala developers.

For applying functional programming:

***Practical FP in Scala: A hands-on approach* by Gabriel Volpe [4]**

A practical book aimed for those familiar with functional programming in Scala who are yet not confident about architecting an application from scratch.

***Functional and Reactive Domain Modeling* by Debasish Ghosh [5]**

Functional and Reactive Domain Modeling teaches you how to think of the domain model in terms of pure functions and how to compose them to build larger abstractions.

Cats Effect version

This book is based on Cats Effect version 3, also known as "CE3".[3]

Most of the concepts in this book apply to the previous major version, Cats Effect 2, but there are a few important differences. The migration guide[4] on the Typelevel Cats Effect project page has a complete listing of what has changed.

Source code for examples and exercises

We believe in learning by doing. Every section of the book includes exercises for you to play with, experiment with, and explore. These are available from GitHub at

https://github.com/inner-product/essential-effects-code

Solutions to the exercises are available on a branch of the above repository, along with being presented both in the text and in an appendix.

In this book you'll often see the ??? method used to mean "it doesn't matter what the implementation is" (for the examples), or "the reader should provide the implementation" (for the exercises). The ??? method is defined in the Scala standard library; it allows the code to compile but will throw an exception at runtime.

Prerequisites

Essential Effects builds on a common set of functional programming techniques: functors, applicatives, and monads. If any are unfamiliar, please review them below. You may not know the technical terms themselves, but you may already know the concepts and have already used them in your own projects.

A deeper dive into functional programming basics can be found in the *Essential Scala* [2] and *Scala with Cats* [3] books.

Functors

A **functor** captures the notion of something you can `map` over, changing its

"contents" (or output) but not the structure itself.

Many types allow you to `map` over them. For example, these types are all functors:

```
List(1, 2, 3).map(_ + 1) // List(2, 3, 4)
    Option(1).map(_ + 1) // Some(2)
    Future(1).map(_ + 1) // ...eventually Future(2)
```

The signature of `map` for some value of type `F[A]`—where type `F` could be `List`, `Option`, etc.—looks like:

```
def map[B](f: A => B): F[B]
```

In *Essential Effects*, we'll be using `map` quite often. Besides `map`, we'll also be using the `as` and `void` extension methods from `Functor`:

In this book we'll display changes to code as a "diff" you might see in a code review. The original code is rendered in red and prefixed with -, and the updated version is in green and prefixed with +.

```
import cats.syntax.all._ ①

val fa: F[A] = ???

- val replaced: F[String] = fa.map(_ => "replacement")
+ val replaced: F[String] = fa.as("replacement") ②

- val voided: F[Unit] = fa.map(_ => ())
+ val voided: F[Unit] = fa.void ③
```

① Import into scope the necessary implicit values and extension methods that use them.

② `as` ignores the value produced by the `Functor` and replaces it with a provided value.

③ `void` also ignores the value produced by the `Functor`, and replaces it with `()`.

Applicatives

An **applicative functor**, also known as applicative, is a functor that can transform *multiple* structures, not just one. Let's start our example by first applying `map` to one `Option` value (it's a *functor*) and extend it to demonstrate the applicative `mapN` method acting on *tuples* of values:

```
                      Option(1).map(_ + 1)          // Some(2) ①
           (Option(1), Option(2)).mapN(_ + _ + 1)   // Some(4) ②
(Option(1), Option(2), Option(3)).mapN(_ + _ + _ + 1) // Some(7) ③
             ...                 ...                //   ...   ④
```

① `map` transforms *one* `Option` → one `Option`.

② `mapN` transforms *two* `Options` → one `Option`.

③ `mapN` transforms *three* `Options` → one `Option`.

④ ... and so on.

More generally, for some applicative type named `F[_]` we can compose a tuple of `F` values into a single `F` value using `mapN`:

```
def map[B](A => B):          F[B] ①
def mapN[C]((A, B) => C):    F[C] ②
def mapN[D]((A, B, C) => D): F[D] ③
    ...                      ...
def mapN[Z]((A, ...) => Z):  F[Z] ④
```

① `map` transforms *one* `F` → one `F`, given a *one*-argument function `A ⇒ B`.

② `mapN` transforms *two* `Fs` → one `F`, given a *two*-argument function `(A, B) ⇒ C`.

③ `mapN` transforms *three* `Fs` → one `F`, given a *three*-argument function `(A, B, C) ⇒ D`.

④ `mapN` transforms *n* `Fs` → one `F`, given an *n*-argument function `(A, ⋯) ⇒ Z`.

In *Essential Effects* we will use applicative methods to compose multiple, independent effects, such as during parallel computation.

In particular, we will often use the symbolic applicative method *> to compose two effects but discard the output of the first. It is equivalent to the following call to `mapN`:

```
import cats.syntax.all._

val first: F[A] = ???
val second: F[B] = ???
```

```
- val third: F[B] = (first, second).mapN((_, b) => b)
+ val third: F[B] = first *> second ①
```

① The *> method composes two effects, `first` and `second`, via `mapN`. If both effects succeed, we ignore the first effect's value, only returning the second effect's value.

Monads

A **monad** is a mechanism for *sequencing* computations: *this* computation happens after *that* computation. Roughly speaking, a monad provides a `flatMap` method for a value `F[A]`:

```
def flatMap[B](f: A => F[B]): F[B]
```

We can use the `flatMap` of some monad `F[_]` to sequence computations:

```
import cats.syntax.all._

val fa: F[A] = ???
def next(a: A): F[B] = ??? ①

val fb: F[B] = fa.flatMap(next)
```

① Produces a new `F[B]` computation from a (pure) value.

Because nested `flatMap` calls can get difficult to read when we have more than two computations to sequence, we can use a `for`-comprehension instead. It is merely syntactic sugar for the nested `flatMap` calls:

```
val fa: F[A] = ???
def nextB(a: A): F[B] = ???
def nextC(b: B): F[C] = ???

val fc: F[C] =
- fa.flatMap { a =>
-   nextB(a).flatMap { b =>
-     nextC(b)
-   }
- }
+ for {
+   a <- fa
+   b <- nextB(a)
+   c <- nextC(b)
+ } yield c
```

[1] https://typelevel.org/cats-effect

[2] [_abstracting_effects_with_typeclasses] details the full set of typeclasses, along with a guide and rationale for translating the concrete effect type to an abstract one.

[3] The dependency "coordinates" for Cats Effect using the sbt build system would be `"org.typelevel" %% "cats-effect" % "3.5.3"`.

[4] https://typelevel.org/cats-effect/docs/migration-guide

Chapter 1. Effects: evaluation and execution

We often use the term *effect* when talking about the behavior of our code, like "What is the *effect* of that operation?" or, when debugging, "Doing that shouldn't have an *effect*, what's going on?", where "what's going on?" is most likely replaced with an expletive. But what is an effect? Can we talk about effects in precise ways, in order to write better programs that we can better understand?

To explore what effects are, and how we can leverage them, we'll distinguish two aspects of code: computing values and interacting with the environment. At the same time, we'll talk about how transparent, or not, our code can be in describing these aspects, and what we as programmers can do about it.

1.1. Computing values: evaluation via substitution

Let's start with the first aspect, computing values. As programmers we write some code, say a method, and it computes a value that gets returned to the caller of that method:

```
def plusOne(i: Int): Int = ①
  i + 1

val x = plusOne(plusOne(12)) ②
```

Here are some of the things we can say about this code:

① `plusOne` is a method that takes an `Int` argument and produces an `Int` value. We often talk about the type signature, or just signature, of a method. `plusOne` has the type signature `Int ⇒ Int`, pronounced "`Int` *to* `Int`" or "`plusOne` *is a function from* `Int` *to* `Int`".

② `x` is a *value*. It is defined as the result of *evaluating* the *expression* `plusOne(plusOne(12))`.

Let's use *substitution* to evaluate this code. We start with the expression `plusOne(plusOne(12))` and substitute each (sub-)expression with its definition, recursively repeating until there are no more sub-expressions:

We're displaying the substitution process as a "diff" you might see in a code review. The original expression is red and prefixed with -, and the result of substitution is in green and prefixed with +.

1. Replace the inner `plusOne(12)` with its definition:

```
- val x = plusOne(plusOne(12))
+ val x = plusOne(12 + 1)
```

2. Replace `12 + 1` with `13`:

```
- val x = plusOne(12 + 1))
+ val x = plusOne(13)
```

3. Replace `plusOne(13)` with its definition:

```
- val x = plusOne(13)
+ val x = 13 + 1
```

4. Replace `13 + 1` with `14`:

```
- val x = 13 + 1
+ val x = 14
```

It is important to notice some particular properties of this example:

1. To understand what `plusOne` does, you *don't* have to look anywhere except the (literal) definition of `plusOne`. There are no references to anything outside of it. This is sometimes referred to as *local reasoning*.
2. Under substitution, programs mean the same thing if they evaluate to the same value. `13 + 1` *means* exactly the same thing as `14`. So does `plusOne(12 + 1)`, or even `(12 + 1) + 1`. This is known as *referential transparency*.

To quote myself while teaching an introductory course on functional programming, "[substitution] is so stupid, even a computer can do it!". It would be fantastic if all programs were as self-contained as `plusOne`, so we humans could use substitution to evaluate code and produce the same value that the computer does.

But substitution is only a *model* of how actual evaluation occurs. It doesn't handle every kind of expression. When does substitution break down? Can you think of some examples?

Here are a few you might have thought of:

1. *When printing to the console.*

   The `println` function prints a string to the console, and has the return type `Unit`. If we apply substitution,

   ```
   - val x = println("Hello world!")
   + val x = ()
   ```

   the meaning—the *effect*—of the first expression is very different from the second expression. Nothing is printed in the latter. Using substitution doesn't do what we intend.

2. *When reading values from the outside world.*

   If we apply substitution,

   ```
   - val name = readLine
   + val name = <whatever you typed in the console>
   ```

   `name` evaluates to whatever *particular* string was read from the console, but that particular string is *not* the same as the evaluation of the expression `readLine`. The expression `readLine` could evaluate to something else.

3. *When expressions refer to mutable variables.*

   If we interact with mutable variables, the value of an expression depends on any possible change to the variable. In the following example, if any code changes the value of `i`, then that would change the evaluation of `x` as well.

   ```
   var i = 12

   - val x = { i += 1; i }
   + val x = 13
   ```

   This example is very similar to the previous one: you could consider typing into the console as writing into a mutable variable whose contents `readLine` returns.

1.2. Interacting with the environment: dealing with side effects

The second aspect of effects, after computing values, is interacting with the environment. And as we've seen, this can break substitution. Environments can change, they are non-deterministic, so expressions involving them do not necessarily evaluate to the same value. If we use mutable state, if we perform hidden side effects—if we break substitution—is all lost? Not at all.

One way we can maintain the ability to reason about code is to localize the "impure" code that breaks substitution. To the outside world, the code will look—and evaluate—as if substitution is taking place. But inside the boundary, there be dragons:

```
def sum(ints: List[Int]): Int = {
  var sum = 0 ①

  ints.foreach(i => sum += i)

  sum
}

sum(List(1, 2, 3)) ②
```

① We've used a mutable variable. The horrors! But nothing outside of `sum` can ever affect it. Its existence is localized to a single invocation.

② When we evaluate the expression that uses `sum`, we get a deterministic answer. Substitution works at this level.

We've optimized, in a debatable way, code to compute the sum of a list, so instead of using an immutable fold over the list we're updating a local variable. From the caller's point to view, substitution is maintained. Within the impure code, we can't leverage the reasoning that substitution gives us, so to prove to ourselves the code behaved we'd have to use other techniques that are outside the scope of this book.

Localization is a nice trick, but won't work for everything that breaks substitution. We need side effects to actually do something in our programs, but side effects are unsafe! What can we do?

1.3. The Effect Pattern

If we impose some conditions, we can tame the side effects into something safer; we'll call these **effects**. There are two parts:

1. ***The type of the program should tell us what kind of effects the program will perform, in addition to the type of the value it will produce.***

   One problem with impure code is we can't *see* that it is impure! From the outside it looks like a method or block of code. By giving the effect a type we can distinguish it from other code. At the same time, we continue to track the type of the result of the computation.

2. ***If the behavior we want relies upon some externally-visible side effect, we separate describing the effects we want to happen from actually making them happen. We can freely substitute the description of effects until the***

point we run them.

This idea is exactly the same as the localization idea, except that instead of performing the side effect at the *innermost* layer of code and hiding it from the outer layers, we delay the side effect so it executes *outside* of any evaluation, ensuring substitution still holds within.

We'll call these conditions the *Effect Pattern*, and apply it to studying and describing the effects we use every day, and to new kinds of effects.

Effect Pattern Checklist

1. Does the type of the program tell us
   a. what **kind of effects** the program will perform; and
   b. what **type of value** it will produce?
2. When externally-visible side effects are required, is the effect description **separate** from the execution?

What effects can you think of? Do they satisify both rules? What makes you sure?

Let's analyze two commonly-used types, `Option` and `Future`, according to the Effect Pattern criteria. Are they effects? Are side effects present, and are they safely managed?

1.3.1. Example: Is `Option` an effect?

Many languages, including Scala, allow the use of the `null` value to mean a value is missing. The programmer (you!) is then required to check if a value is `null` or not, otherwise the dreaded `NullPointerException` is thrown at runtime.

```
def isValid(filename: String) =
  filename.length > 0 &&   ①
  filename.startsWith("/") ①

isValid(null)
```

① Beware, `NullPointerException`'s abound.

To prevent us from forgetting to check which case we are in, Scala offers another way to encode optionality, as an *algebraic data type*:

```
sealed trait Option[+A]
```

```
case class Some[A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

Is `Option[A]` an effect? Let's check the criteria:

1. *Does* `Option[A]` *tell us what kind of effects the program will perform, in addition to the type of the value it will produce?*

   Yes: if we have a value of type `Option[A]`, we know the effect is optionality from the name `Option`, and we know it may produce a value of type `A` from the type parameter `A`.

2. *Are externally-visible side effects required?*

   Not really. The `Option` data type is an interface representing optionality that maintains substitution. We can replace a method call with its implementation and the meaning of the program won't change.

   There is one exception—pun intended—where an externally-visible side effect might occur:

   ```
   def get(): A =
     this match {
       case Some(a) => a
       case None => throw new NoSuchElementException("None.get")
     }
   ```

   Calling `get` on a `None` is a programmer error, and raises an exception which in turn may result in a stack trace being printed. However this side effect is not core to the concept of exceptions, it is just the implementation of the default exception handler. The essence of exceptions is non-local control flow: a jump to an exception handler in the dynamic scope, which together is not an externally-visible side effect.

With these two criteria satisfied, we can say **yes**, `Option[A]` is an effect!

It may seem strange to call `Option` an effect since it doesn't perform any side effects. The point of the first condition of the Effect Pattern is that the type should make the presence of an effect *visible*. As we mentioned, the traditional alternative to `Option` would be to use a `null` value, but then how could you tell that a value of type `A` could be `null` or not? Some types which could have a `null` value are not intended to have the concept of a missing value. `Option` makes this distinction apparent.

Effect Pattern Checklist: `Option[A]`

1. Does the type of the program tell us
   a. what **kind of effects** the program will perform; and

      ✔ The `Option` type represents *optionality*.

      Optionality means a value may (or may not) exist.

   b. what **type of value** it will produce?

      ✔ A value of type `A`, if one exists.

2. When externally-visible side effects are required, is the effect description **separate** from the execution?

   ✔ No externally-visible side effects are required.

✔ Therefore, `Option` is an effect.

1.3.2. Example: Is `Future` an effect?

`Future` is known to have issues that aren't easily seen. For example, look at this code, where we reference the same `Future` to run it twice:

```
val print = Future(println("Hello World!"))
val twice =
  print
    .flatMap(_ => print)
```

What output is produced?

```
Hello World!
```

It is only printed once! Why is that?

The reason is that the `Future` is scheduled to be run immediately upon construction. So the side effect will happen (almost) immediately, even when other "descriptive" operations—the subsequent `print` in the `flatMap`—happen later. That

is, we describe performing `print` twice, but the side effect is only executed once!

Compare this to what happens when we substitute the definition of `print` into `twice`:

1. Replace the first reference to `print` with its definition:

```
  val print = Future(println("Hello World!"))
  val twice =
-   print
+   Future(println("Hello World!"))
      .flatMap(_ => print)
```

2. Replace the second reference to `print` with its definition, and remove the definition of `print` since it has been inlined.

```
- val print = Future(println("Hello World!"))
  val twice =
    Future(println("Hello World!"))
-     .flatMap(_ => print)
+     .flatMap(_ => Future(println("Hello World!")))
```

We now have:

```
val twice =
  Future(println("Hello World!"))
    .flatMap(_ => Future(println("Hello World!")))
```

Running it, we then see:

```
Hello World!
Hello World!
```

This is why we say `Future` is *not* an effect: the substitution of expressions with their definitions doesn't have the same meaning.

Effect Pattern Checklist: `Future[A]`

1. Does the type of the program tell us
   a. what **kind of effects** the program will perform; and

✔ A `Future` represents an *asynchronous computation*.

b. what **type of value** it will produce?

> ✔ A value of type `A`, if the asynchronous computation is successful.

2. When externally-visible side effects are required, is the effect description **separate** from the execution?

> ✔ Externally-visible side effects *are required*: the body of a `Future` can do anything, including side effects.
>
> ✖ But those side effects are *not* executed after the description of composed operations; the execution is scheduled immediately upon construction.

✖ Therefore, `Future` does not separate effect description from execution: it is *unsafe*.

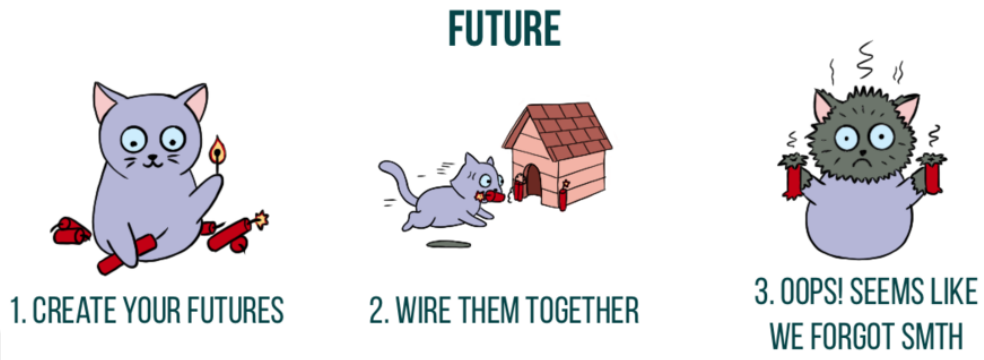

Figure 1. `Future` is unsafe. Image by @impurepics.

1.4. Capturing arbitrary side effects as an effect

We've seen the `Option` effect type, which doesn't involve side effects, and we've examined why `Future` isn't an effect. So what about an effect that does involve side effects, but safely?

This is the purpose of the `IO` effect type in `cats.effect`. It is a data type that allows us to capture *any* side effect, but in a safe way, following our Effect Pattern. We'll first build our own version of `IO` to understand how it works.

Let's create our first effect: we want to capture *arbitrary* side effects. We'll

demonstrate it by describing an effect to print a string to the console, then subsequently execute it.

Example 1. Capturing side effects with the `MyIO` effect type.

```
package com.innerproduct.ee.effects

case class MyIO[A](unsafeRun: () => A) ①

object MyIO {
  def putStr(s: => String): MyIO[Unit] =
    MyIO(() => println(s)) ②
}

object Printing extends App {
  val hello = MyIO.putStr("hello!") ③

  hello.unsafeRun() ④
}
```

① The side effect we want to delay is captured as the function `unsafeRun`. We named it `unsafeRun` because we want to let everyone know this function does not maintain substitution.

② Our printing effect `putStr` is defined by constructing a `MyIO` value that delays the execution of the `println` function.

③ We *describe* the printing of `"hello!"` as a `MyIO` value. But it hasn't been executed yet.

④ Here we *explicitly* run the effect.

If we run the `Printing` program it outputs:

```
hello!
```

For the `Printing` program, let's check that `MyIO` maintains substitution, by replacing each expression with its definition, recursively.

1. Our original program:

```
object Printing extends App {
  val hello = MyIO.putStr("hello!")

  hello.unsafeRun()
}
```

2. Replace the value `hello` with its definition `MyIO.putStr("hello!")`:

```
- val hello = MyIO.putStr("hello!")
-
- hello.unsafeRun()
+ MyIO.putStr("hello!")
+   .unsafeRun()
```

3. Replace the `MyIO.putStr` expression with its definition `MyIO(() => println("hello"))`:

```
- MyIO.putStr("hello!")
+ MyIO(() => println("hello!"))
    .unsafeRun()
```

4. Replace the `unsafeRun()` expression with its definition, which evaluates `unsafeRun` itself:

```
- MyIO(() => println("hello!"))
-   .unsafeRun()
+ println("hello!")
```

After recursively replacing the program's expressions with its definitions, the body of the `Printing` program is equivalent to the expression `println("hello!")`. So, **yes**, `MyIO` maintains substitution: after every evaluation, the meaning of the program is preserved.

1.5. Composing effects

We can construct individual effects, and run them, but how do we combine them? We may want to modify the output of an effect (via `map`), or use the output of an effect to create a new effect (via `flatMap`). Let's add these methods to our `MyIO`.

But be careful! Composing effects must not execute them. We require composition to maintain substitution, so we may build effects out of other effects.

Example 2. Adding `map` *and* `flatMap` *methods to* `MyIO`.

```
package com.innerproduct.ee.effects

case class MyIO[A](unsafeRun: () => A) {
  def map[B](f: A => B): MyIO[B] =
    MyIO(() => f(unsafeRun())) ①
```

```
  def flatMap[B](f: A => MyIO[B]): MyIO[B] =
    MyIO(() => f(unsafeRun()).unsafeRun()) ②
}

object MyIO {
  def putStr(s: => String): MyIO[Unit] =
    MyIO(() => println(s))
}

object Printing extends App {
  val hello = MyIO.putStr("hello!")
  val world = MyIO.putStr("world!")

  val helloWorld: MyIO[Unit] = ③
    for {
      _ <- hello
      _ <- world
    } yield ()

  helloWorld.unsafeRun()
}
```

① The definition of `map` is straightforward: We create a new `MyIO` that must return a value of type `B`. How can we get a `B`? We have the `A ⇒ B` function `f`, so where can we get an `A` value? We use `unsafeRun`.

② The definition of `flatMap` is slightly more complicated. Again we create a new `MyIO` that must return a `B`. We call `f` with the output of `unsafeRun`, but this gives us a `MyIO[B]`, not a `B`. But if we invoke `unsafeRun` on *that* `MyIO`, it will produce the `B` value we need.

This definition agrees with what `flatMap` is supposed to do: it *sequences* two operations, where one happens before the other.

③ We combine the `hello` and `world` effects using a `for`-comprehension (which uses `flatMap`), and their composition returns a single effect.

Running the `Printing` program produces:

```
hello!
world!
```

Exercise 1: Timing

Code available at `effects/Timing.scala`.

```
package com.innerproduct.ee.effects

import scala.concurrent.duration.FiniteDuration

object Timing extends App {
  val clock: MyIO[Long] =
    ??? ①

  def time[A](action: MyIO[A]): MyIO[(FiniteDuration, A)] =
    ??? ②

  val timedHello = Timing.time(MyIO.putStr("hello"))

  timedHello.unsafeRun() match {
    case (duration, _) => println(s"'hello' took $duration")
  }
}
```

① Write a clock action that returns the current time in milliseconds, i.e., via `System.currentTimeMillis`.

② Write a timer that records the duration of another action.

Solution to Exercise

1.5.1. MyIO as an effect

Let's check `MyIO` against our Effect Pattern:

Effect Pattern Checklist: `MyIO[A]`

1. Does the type of the program tell us
   a. what **kind of effects** the program will perform; and

      ✔ A `MyIO` represents a (possibly) *side effecting computation*.

   b. what **type of value** it will produce?

✔ A value of type `A`, if the computation is successful.

2. When externally-visible side effects are required, is the effect description **separate** from the execution?

   ✔ Externally-visible side effects *are required*: when executed, a `MyIO` can do anything, including side effects.

   ✔ We describe `MyIO` values by constructing them and by composing with `map` and `flatMap`. The execution of the effect only happens when `unsafeRun` is called.

✔ Therefore, `MyIO` is an effect!

By satisfying the Effect Pattern we know our `MyIO` effect type is safe to use, even when programming with side effects. At any point before we invoke `unsafeRun` we can rely on substitution, and therefore we can replace any expression with its value—and vice-versa—to safely refactor our code.

In the next chapter we'll introduce the `cats.effect.IO` type, which is built using the same techniques as our simpler `MyIO` type.

What's a "thunk"?

While we don't use the term in this book, you might see a reference to the term "thunk" while reading about functional programming, programming with effects, or a host of other subjects. For example, you might see a phrase like "pass a *thunk* as the first argument to the method." What's a "thunk"?

A **thunk** is simply a delayed computation. The name is a pun on the past tense of "think", so the value of the thunk is available after the "thinking" of the computation is complete.[1] A thunk may optionally memoize its result, avoiding recomputation when subsequently evaluated.

You've most likely encountered a thunk in Scala as a *call-by-name* parameter:

```
def doSomething[A](thunk: => A) ①
```

① Whenever it is evaluated, `thunk` produces a value of type `A`.[2]

Call-by-name parameters can't themselves be values, so a thunk can alternatively have the type signature `() ⇒ A`: a zero-argument function that

produces a value of type `A` when evaluated. For example, we used this form in our `MyIO` data type, where it contained a thunk named `unsafeRun`, because we require effects delay their execution:

```
case class MyIO[A](unsafeRun: () => A)
```

Sometimes the terminology gets blurred a bit, where someone might say a value of type `MyIO` *is* a thunk, since you can use it to produce a delayed computation, rather than the more literal interpretation of it *having* a thunk. Both interpretations can be useful.

1.6. Summary

1. The substitution model of evaluation gives us local reasoning and fearless refactoring.
2. Interacting with the environment can break substitution. One solution is to localize these side effects so they don't affect evaluation.
3. Another solution is the Effect Pattern: a set of conditions that makes the presence of effects more visible while ensuring substitution is maintained. An effect's type tells us what kind of effects the program will perform, in addition to the type of the value it will produce. Effects separate describing what we want to happen from actually making it happen. We can freely substitute the description of effects up until the point we run them.
4. We demonstrated a way to safely capture side effects via the `MyIO[A]` type, which delayed the side effect until the `unsafeRun` method is called. We produced new `MyIO` values with the `map` and `flatMap` combinators.

[1] https://en.wikipedia.org/wiki/Thunk

[2] That is, call-by-name parameters are *not* memoized.

References

- [1] Dave Gurnell and Noel Welsh. https://creativescala.org
- [2] Noel Welsh and Dave Gurnell. *Essential Scala*. https://underscore.io/books/essential-scala
- [3] Noel Welsh and Dave Gurnell. *Scala with Cats*. https://www.scalawithcats.com
- [4] Gabriel Volpe. *Practical FP in Scala: A hands-on approach*. https://leanpub.com/pfp-scala
- [5] Debasish Ghosh. *Functional and Reactive Domain Modeling*. https://www.manning.com/books/functional-and-reactive-domain-modeling
- [6] Allen B. Downey. *The Little Book of Semaphores*. https://greenteapress.com/wp/semaphores