ESSENTIAL EFFECTS

ADAM ROSIEN

Essential Effects

Adam Rosien

Essential Effects

Version 3988131 @ 2021-07-05.

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 chatroom.*

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, large or small, to: adam+effects@inner-product.com.

Contents

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 chatroom,[1] 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 to the community.

Thank you to all the reviewers and proofreaders: Charles Adetiloye, Roman Arkharov, Samir Bajaj, Gavin Bisesi, Guillaume Bogard, Bogdan C., Christopher Davenport, Francisco Diaz, Saskia Gennrich, Debasish Ghosh, Daniel Hinojosa, Matt Hughes, Lars Hupel, Marc Ramírez Invernon, Sándor Kelemen, Jakub Kozłowski, Fabio Labella, Chris Lan, Zachary McCoy, Juan Mendez, 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[2] 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.[3]

A functional programming curriculum

While there are many excellent books focusing on functional programming in Scala, we specifically recommend the following books as a *functional programming curriculum* 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 versions

This book is based on both Cats Effect version 2.2.0 ("CE2")[4] and Cats Effect 3 ("CE3"). In this book if there is a necessary difference between the Cats Effect 2 and 3 code the former will be annotated with "CE2" and the latter will be annotated with "CE3".[5]

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

In the exercises you'll often see the ??? method used to mean "it doesn't matter what the implementation is" (for examples), or "the reader should provide the implementation" (for exercises). The ??? method is defined in the Scala standard library and will throw an exception at runtime to denote something is unimplemented.

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.

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.implicits._ 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* `F`s → one `F`, given a *two*-argument function `(A, B) ⇒ C`.

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

④ `mapN` transforms *n* `F`s → 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.implicits._

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.implicits._

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://gitter.im/typelevel/cats-effect

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

[3] Appendix B, *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.

[4] The dependency "coordinates" for Cats Effect using the sbt build system would be `"org.typelevel" %% "cats-effect" % "2.2.0"` or `"org.typelevel" %% "cats-effect" % "3.0.1"`, respectively.

[5] Appendix Appendix C, *Changes in Cats Effect 3* contains a detailed list of differences between Cats Effect 2 and 3.

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. The substitution model of evaluation

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. 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*.

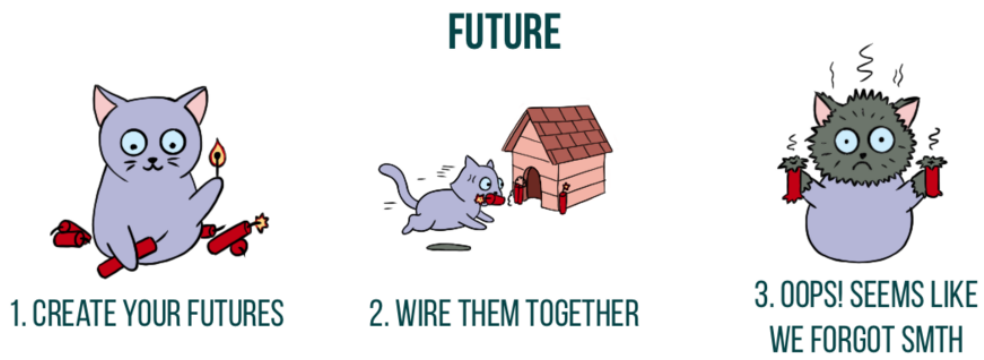

Figure 1. `Future` *is unsafe. Image by @impurepics.*

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.[6] 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`.[7]

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.

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

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

Chapter 2. Cats Effect `IO`

We built our own `MyIO` effect type to understand how to delay side effects, while maintaining the ability to compose new behaviors using methods like `map` and `flatMap`. We'll now introduce the `cats.effect.IO` type, which has the same properties, and briefly introduce the typical ways of constructing, transforming and executing `IO` values. We'll also show how to build applications using effects with `cats.effect.IOApp`.

2.1. Constructing `IO` values

Most often we'll use `IO.delay` to capture a side effect as an `IO` value:

```
def delay[A](a: => A): IO[A]
```

`IO.delay` takes a call-by-name (lazily-evaluated) argument, delaying the evaluation of the code:

```
val hw: IO[Unit] = IO.delay(println("Hello world!")) ①
```

① Delay the evaluation of this expression so that when it is (later) executed, it prints `Hello world` to the console, and produces the value `()`.

`IO.apply` is an alias for `IO.delay`. This lets us write `IO(…)` instead of `IO.delay(…)`, which is shorter:

```
- val hw: IO[Unit] = IO.delay(println("Hello world!"))
+ val hw: IO[Unit] = IO(println("Hello world!"))
```

When the effect is executed, what happens if the side effect throws an exception? For example, what happens if you create an `IO` value like this?

```
val ohNoes: IO[Int] =
  IO.delay(throw new RuntimeException("oh noes!"))
```

We know that even though this effect has type `IO[Int]`, it can't produce an `Int` because of the `throw`. However, the `throw` side effect is delayed until the `IO` is executed, and then and only then it will throw the exception.

Although it is less common, we can also construct `IO` values from existing "pure" values:

```
val twelve: IO[Int] = IO.pure(12)
```

Be careful! Do not perform any side effects when calling `IO.pure`, because they will be eagerly evaluated and that will break substitution. If you aren't sure, use `IO.delay` or `IO.apply` to be safe.

We can also "lift" an exception into `IO`, as long as we provide the "expected" type of the `IO` had it succeeded, either explicitly or through type inference:

```
val ohNoes: IO[Int] =
  IO.raiseError(new RuntimeException("oh noes!")) ①
```

① Note the difference between the use of `IO.raiseError` and the previous exercise, which *threw* an exception in the call to `IO.delay`. Here we already have the exception as a value.

Since it is a common alternative effect type, there is a general way to transform `scala.concurrent.Future` values into `IO` values:

```
val fut: IO[String] = IO.fromFuture(IO(futurish)) ①

def futurish: Future[String] = ???
```

① We're converting a `Future` to an `IO`, but are required to pass the `Future` inside an `IO`. Why do you think that is?[8]

2.2. Transforming `IO` values

Once we have `IO` values, we can call various methods to produce new `IO` values. These methods are often called *combinators*.

`IO` is a *functor*; we can `map` over it:

```
IO(12).map(_ + 1) ①
```

① When executed, produces 13.

`IO` is an *applicative*; we can `mapN` over two or more values:

```
(IO(12), IO("hi")).mapN((i, s) => s"$s: $i") ①
```

① When executed, produces `"hi: 12"`.

`IO` is a *monad*; we can `flatMap` over it, or more conveniently, we can use a `for` -comprehension:

```
for {
  i <- IO(12)
  j <- IO(i + 1)
} yield j.toString ①
```

① When executed, produces "13".

There are many other combinators available. The Appendix A, *Cheatsheets* appendix details the most common.

2.2.1. Error handling

As we've seen, an `IO` computation can fail, either by throwing an exception during execution, or by capturing an existing exception via `IO.raiseError`. We can, however, detect these failures and do something about it. A common combinator for error handling[9] is `handleErrorWith`, which has a similar signature to `flatMap` except it accepts error values:

```
def handleErrorWith[AA >: A](f: Throwable => IO[AA]): IO[AA]
```

We *handle* the error by producing a new effect:

```
val ohNoes =
  IO.raiseError[Int](new RuntimeException("oh noes!"))

val handled: IO[Int] =
  ohNoes.handleErrorWith(_ => IO(12))
```

If you want to simply provide a successful value, you can use `handleError`:

```
- ohNoes.handleErrorWith(_ => IO(12))
+ ohNoes.handleError(_ => 12)
```

But `handleErrorWith` doesn't need to produce a successful effect, it could produce an effect which itself fails:

```
val handled: IO[Int] =
  ohNoes.handleErrorWith(t => IO.raiseError(new OtherException(t)))
```

If we explicitly want to transform the error into another error, we could use `adaptError` instead:

```
- ohNoes.handleErrorWith(t => IO.raiseError(new OtherException(t)))
+ ohNoes.adaptError(t => new OtherException(t))
```

These combinators *hide* the error-handling logic from the code that consumes these effects: for example a method may return an `IO[Int]` that could have been produced by a `handleErrorWith` call. There is an interesting alternative to this where we instead handle errors by transforming them into `Either` values, so an `IO[A]` now becomes an `IO[Either[Throwable, A]]`:

```
def attempt(): IO[Either[Throwable, A]]
```

Instead of hiding the error-handling we're now *exposing* the error, but also *delaying* the error-handling by "lifting" the error into a (successful) `IO` value:

```
val attempted: IO[Either[Throwable, Int]] =
- ohNoes
-   .map(i => Right(i): Either[Throwable, Int])
-   .handleErrorWith(t => Left(t))
+ ohNoes.attempt
```

Error-handling Decision Tree

If an error occurs in your `IO[A]` do you want to...

perform an effect?

Use `onError(pf: PartialFunction[Throwable, IO[Unit]]): IO[A]`.

transform any error into another error?

Use `adaptError(pf: PartialFunction[Throwable, Throwable]): IO[A]`.

transform any error into a successful value?

Use `handleError(f: Throwable ⇒ A): IO[A]`.

transform *some* kinds of errors into a successful value?

Use `recover(pf: PartialFunction[Throwable, A]): IO[A]`.

transform *some* kinds of errors into another effect?

Use `recoverWith(pf: PartialFunction[Throwable, IO[A]]): IO[A]`.

make errors visible but delay error-handling?

Use `attempt: IO[Either[Throwable, A]]`.

Otherwise, use `handleErrorWith(f: Throwable ⇒ IO[A]): IO[A]`.

2.3. Executing `IO` values

We've delayed any side effects by encapsulating them into an `IO` value. When we're done composing our program we'll finally execute the effects. There are a number of methods to execute them, and they all are prefixed with `unsafe` to denote that side effects will get executed and that our substitution process no longer applies.

The most common `unsafe` method you'll encounter is `unsafeRunSync`. *Run* means execute, and *sync* means *synchronous* execution; together they run the effects synchronously and return the result. Invoking `unsafeRunSync` on an `IO[A]` will produce a value of type `A` if the effect succeeds:

```
def unsafeRunSync: A
```

For example,

```
IO("hello world!").unsafeRunSync
```

produces the value `"hello world!"`. If the effect throws an exception during execution, or if it was created directly with `IO.raiseError`, then that exception would be thrown from `unsafeRunSync`.

There is naturally an `unsafeRunAsync`, and other methods, but they are less frequently used. More common, however, is the `unsafeToFuture` method, which you may use to integrate your effectful code with legacy interfaces that may consume `scala.concurrent.Future`:

```
def unsafeToFuture: Future[A]
```

But again, be warned! As a general rule, you should *not* be invoking any `unsafe` method in your code. When experimenting in the REPL or some other throw-away code, sure. Otherwise, don't do it! Instead, you'll delegate this responsibility to types like `IOApp`, explained below. But first we'll check if `cats.effect.IO` is an effect or not.

2.4. `IO` as an effect

Let's evaluate `IO` against our Effect Pattern checklist.

Effect Pattern Checklist: `IO[A]`

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

      ✔ An `IO` 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, an `IO` can do anything, including side effects.

   ✔ We describe `IO` values by various constructors, and describe compound effects by composing them with methods like `map`, `mapN`, `flatMap`, and so on. The execution of the effect only happens when an `unsafe*` method is called.

✔ Therefore, `IO` is an effect!

Figure 2. `IO` *is safe: it separates effect description from execution. Image by @impurepics.*

2.5. Executing effects in applications with `IOApp`

How do we integrate effects into our applications? We'll use our Effect Pattern again: given the presence of side effects, separate the description of our program

from its execution. Our applications then mirror this structure:

1. We describe the effects we want to happen; and
2. The application executes them when we run it.

To do this Cats Effect provides the `IOApp` type for applications. `IOApp` is an executable Scala type—something that has a `main` method—that requires you to declare your effects as a single `IO` value, and it performs the task of executing those effects.

Example 3. "Hello World" as an `IOApp` *program. Code available at* `resources/HelloWorld.scala`.

```
package com.innerproduct.ee.resources

import cats.effect._

object HelloWorld extends IOApp { ①
  def run(args: List[String]): IO[ExitCode] = ②
    helloWorld.as(ExitCode.Success) ③

  val helloWorld: IO[Unit] =
    IO(println("Hello world!"))
}
```

① Your application should be an `object` and needs to extend `IOApp`.

② The application entry point is the `run` method, which must return `IO[ExitCode]`.

③ Declare the computations that will be run.

The abstract `run` method of `IOApp` requires you to return an `IO` value. How does `IOApp` execute the effects? Since an `IO` only executes if an unsafe method like `unsafeRunSync` is called, `IOApp` must invoke it for us.

Our application's effect produces an `ExitCode`, an enumeration that abstracts over the exit code to return to the operating system when the application ends. Most applications will return `ExitCode.Success`, but you can always return `ExitCode.Error` or construct an `ExitCode` with a numeric value, where non-zero values denote an error.

Let's get some practice creating an `IOApp`: we'll build a ticking clock.

Exercise 2: Ticking Clock

Let's create a more complex application: a ticking clock. It should print the current time every second.

Code available at `io/TickingClock.scala`.

```
package com.innerproduct.ee.io

import cats.effect._

object TickingClock extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    tickingClock.as(ExitCode.Success)

  val tickingClock: IO[Unit] = ??? ①
}
```

① `tickingClock` should print the current time every second. Use `System.currentTimeMillis` to be simple. How do you wake up every second? (Hint: You can use `IO.sleep`.) And how do you do this repeatedly?

Solution to Exercise

Going forward, we'll switch to using `IOApp` in examples and exercises.

2.6. Summary

1. `cats.effect.IO` is an effect that can encapsulate any side effect.
   a. Constructors produce an `IO` from pure values, delayed side effects, errors, and other types like `Future`.
   b. Combinators let you build new effects, transform their outputs, and handle errors. It is essential that every combinator avoids the execution of any effect, otherwise substitution will be broken.
   c. We can execute `IO[A]` values, who produce either a value of type `A` or raise an exception. You should only run them at the very "edges" of your programs via its `unsafe`-prefixed methods.
2. `cats.effect.IOApp` lets you describe your application as a single `IO` effect that it executes.

[8] As you may recall, a `Future` is scheduled for execution during its construction. However, an `IO` effect must delay its execution, so we delay the `Future` by wrapping its creation within its own `IO` value.

[9] These combinators are defined in the `ApplicativeError` and `MonadError` typeclasses of the Cats library, along with their extension methods. An easy way to make the extension methods available is to add `import`

`cats.syntax.all._` to your imports.

Chapter 3. Parallel execution

Capturing side effects as the composable `IO` data type is useful in itself, but once we do that we can go farther. If we have multiple `IO` values that don't depend on each other, shouldn't we be able to run them in parallel? That way our overall computation can become more efficient.

First we'll discuss if `IO` itself supports parallelism, or not. We'll then talk about how `IO` can support parallelism, and how that parallelism is implemented. We'll then see some examples of different ways to compose `IO` values in parallel.

3.1. Does `IO` support parallelism?

To answer the question of whether or not `IO` supports parallelism, let's first compare it to a similar data type, `scala.concurrent.Future`, which we've seen supports parallelism by scheduling work on multiple threads via a `scala.concurrent.ExecutionContext`.

Let's compose multiple `Future` values using both `flatMap` (via a `for`-comprehension) and `mapN`.

In the code below, is the effect of `hw1` the same as the effect of `hw2`? Do `hello` and `world` run in parallel, or not? What output will we see printed to the console?

Example 4. `parallel/Future1.scala`*: Is* `hw1` *the same as* `hw2`*?*

```
package com.innerproduct.ee.parallel

import cats.implicits._
import scala.concurrent._
import scala.concurrent.duration._

object Future1 extends App {
  implicit val ec = ExecutionContext.global

  val hello = Future(println(s"[${Thread.currentThread.getName}] Hello")) ①
  val world = Future(println(s"[${Thread.currentThread.getName}] World")) ①

  val hw1: Future[Unit] =
    for {
      _ <- hello
      _ <- world
    } yield ()

  Await.ready(hw1, 5.seconds) ②

  val hw2: Future[Unit] =
    (hello, world).mapN((_, _) => ())

  Await.ready(hw2, 5.seconds) ③
}
```

① We've added some helper code to show the current thread during the execution of the `Future`.

② Is the effect of `hw1` the same as...

③ ... the effect of `hw2`?

If we run this program, we see the following output (your thread ids may be different):

```
[scala-execution-context-global-10] Hello
[scala-execution-context-global-11] World
```

We see only *one* pair of `Hello` and `World` printed. Why? Because `Future` eagerly schedules the action, and caches the result. This breaks rule #2 of our Effect Pattern: the unsafe side effect is *not* separately executed from functions like the `Future` constructor.

Sorry! We felt the need to remind you that `Future` eagerly executes side effects. Let's ensure that we delay the side effects as long as possible:

Example 5. `parallel/Future2.scala`*: Is* `hw1` *the same as* `hw2`*?*

```
package com.innerproduct.ee.parallel

import cats.implicits._
import scala.concurrent._
import scala.concurrent.duration._

object Future2 extends App {
  implicit val ec = ExecutionContext.global

  def hello = Future(println(s"[${Thread.currentThread.getName}] Hello")) ①
  def world = Future(println(s"[${Thread.currentThread.getName}] World")) ①

  val hw1: Future[Unit] =
    for {
      _ <- hello
      _ <- world
    } yield ()

  Await.ready(hw1, 5.seconds) ②

  val hw2: Future[Unit] =
    (hello, world).mapN((_, _) => ())

  Await.ready(hw2, 5.seconds) ③
}
```

① Note we've changed from a `val` to a `def` to avoid the caching behavior of `Future` from the previous example.

② Is the effect of `hw1` the same as...

③ ... the effect of `hw2`?

If we run this program, we see the following output:

```
[scala-execution-context-global-10] Hello
[scala-execution-context-global-10] World
[scala-execution-context-global-11] World ①
[scala-execution-context-global-10] Hello ①
```

① Notice `World` is printed before `Hello`!

The output now—correctly—shows us *two* pairs of `Hello` and `World` outputs. And we see the second pair of outputs, from calling `mapN`, running on different threads.

But be careful! Even though we see output happening on two different threads, that *doesn't* imply that those computations happened in parallel. How might you

be able to show they ran in parallel, or not? (It's not too important to answer this question.)

At the same time, `hw2` computation is actually non-deterministic, so you may also see `Hello` printed before `World`. What makes the output non-deterministic?

While the effects of `hw2` are non-deterministic, we can say one thing about the execution of `hw1`, the computation that uses a `for`-comprehension. Because a `for`-comprehension is syntactic sugar for a series of nested `flatMap` calls, we can say with certainty that the effect of `world` will *always* execute after the effect of `hello`, because the `world` effect is only created (and subsequently executed) once the value of `hello` is computed:

```
val hw1: Future[Unit] =
- for {
-   _ <- hello
-   _ <- world
- } yield ()
+ hello.flatMap { _ => ①
+   world.map { _ =>
+     ()
+   }
+ }
```

① A `for`-comprehension is syntactic sugar a series of nested `flatMap` calls. Once we desugar the syntax we can see `world` only executes after the result of `hello` is computed, because the result of `hello` is passed to the function given to the `flatMap`.

On the other hand, we can infer that `hw2`, which uses `mapN`, would run `hello` and `world` in parallel. Why is that?

If we partially evaluate the definition of `hw2` a little bit, we can see something important going on:

```
val hw2: Future[Unit] =
  (
-   hello,
+   Future(println(s"[${Thread.currentThread.getName}] Hello")) ①
-   world
+   Future(println(s"[${Thread.currentThread.getName}] World")) ②
  ).mapN(((_, _) => ())
```

① We know when we construct a `Future`, it gets scheduled.

② The second `Future` is also scheduled, so now both will be executed independently.

This demonstrates that for `Future`, `flatMap` and `mapN` have *different* effects with respect to parallelism.

But note: it *isn't* the case `mapN` for `Future` is implemented with parallelism but `flatMap` is implemented as something sequential. The parallelism comes as a side effect—pun intended—of `Future` eagerly scheduling the computation, which happens *before* `mapN` itself is evaluated.

What about `IO`? Does using `mapN` vs. `flatMap` have a different effect, like `Future` does?

Example 6. `parallel/IOComposition.scala`*: Is* `hw1` *the same as* `hw2`*?*

```
package com.innerproduct.ee.parallel

import cats.effect._
import cats.implicits._

object IOComposition extends App {
  val hello = IO(println(s"[${Thread.currentThread.getName}] Hello")) ①
  val world = IO(println(s"[${Thread.currentThread.getName}] World"))

  val hw1: IO[Unit] =
    for {
      _ <- hello
      _ <- world
    } yield ()

  val hw2: IO[Unit] =
    (hello, world).mapN((_, _) => ())

  hw1.unsafeRunSync() ②
  hw2.unsafeRunSync() ③
}
```

① We're using code equivalent to the previous examples with `Future` in order to be as consistent as possible.

② Is the effect of `hw1` the same as...

③ ... the effect of `hw2`?

The `IOComposition` program outputs:

```
[main] Hello
[main] World
[main] Hello
[main] World
```

We now see the expected output, but all the threads are the same. Did you expect that?

Do you think `hw2` has non-deterministic output, like the previous example using `Future`?

`IO` doesn't provide any support for the effect of parallelism! And this is by design, because we want different effects to have different types, as per our Effect Pattern. [10]

3.2. The `Parallel` typeclass

As we've seen, unlike `Future`, `IO` itself *doesn't* provide any support for parallelism. So how can we achieve it?

We will again follow our Effect Pattern and apply rule #1: *the type should reflect the effect*. If `IO` doesn't support parallelism, we need a *new* type that does. In `cats.effect`, this type is named `IO.Par` (`Par` for "parallel").

```
sealed abstract class IO[+A] { ... } ①

object IO {
  class Par[+A] { ... } ②

  object Par {
    def apply[A](ioa: IO[A]): Par[A] = ??? ③
    def unwrap[A](pa: Par[A]): IO[A] = ??? ③
  }
}
```

① The (sequential) `IO` data type.

② `IO`'s parallel data type, `IO.Par`.

③ Methods to transform between `IO` and `IO.Par` values.

`IO.Par` will *not* have a `Monad` instance, because we do not want to be able to serialize the execution of multiple actions. Instead it will have an `Applicative` instance, to compose independent `IO.Par` values:

```
implicit def ap(implicit cs: ContextShift[IO]): Applicative[IO.Par] = ①
  new Applicative[IO.Par] {
    def pure[A](a: A): IO.Par[A] = IO.Par(IO.pure(a))
    def map[A, B](pa: IO.Par[A])(f: A => B): IO.Par[B] = ???
    def product[A, B](pa: IO.Par[A], pb: IO.Par[B]): IO.Par[(A, B)] = ??? ②
  }
```

① We require a `ContextShift[IO]` to be able to switch computations to different threads. We'll talk more about `ContextShift` in Chapter 5, *Shifting contexts*, but for the present you can think of it as something similiar to a `scala.concurrent.ExecutionContext` or thread pool.

② The implementation of `product` will ensure that `pa` and `pb` execute on different threads, using `cs`.

It's a bit verbose to have to switch types when we translate between sequential and parallel execution. It would look like:

Example 7. Switching between sequential (`IO`) and parallel (`IO.Par`) types.

```
val ia: IO[A] = IO(???)
val ib: IO[B] = IO(???)

def f(a: A, b: B): C = ???

val ipa: IO.Par[A] = IO.Par(ia) ①
val ipb: IO.Par[B] = IO.Par(ib) ①

val ipc: IO.Par[C] = (ipa, ipb).mapN(f) ②

val ic: IO[C] = IO.Par.unwrap(ipc) ③
```

① Translate each `IO` to `IO.par`.

② Compose, in parallel, two `IO.par` into one `IO.Par` via `mapN`.

③ Translate `IO.Par` back to `IO`.

The `Parallel` typeclass from the Cats library (*not* Cats Effect) captures the concept of translating between two related data types:

```
trait Parallel[S[_]] { ①
  type P[_] ②

  def monad: Monad[S] ③

  def applicative: Applicative[P] ④

  def sequential: P ~> S ⑤

  def parallel: S ~> P   ⑤
}
```

① Typeclass instances are about the type `S` (for *sequential*). For example, there will be a typeclass instance `Parallel[IO]`, where `IO` is the sequential type to be transformed.

② The typeclass instance defines the `P` type (for *parallel*). For the `Parallel[IO]` typeclass instance, `P` would be `IO.Par`.

③ `S` must have a `Monad`. That is, operations using `S` must be sequenced.

④ `P` must have an `Applicative`. That is, operations using `P` must not have any data ordering dependencies.

⑤ A `Parallel` instance must be able to transform from sequential values to parallel values, and back.

The `~>` symbol is a type alias for `cats.arrow.FunctionK`, which is a transformation from some type `F[A]` to another type `G[A]`, for any type `A`. So the type `P ~> S` is equivalent to code like `def apply[A](pa: P[A]): S[A]`.

Diagrammatically:

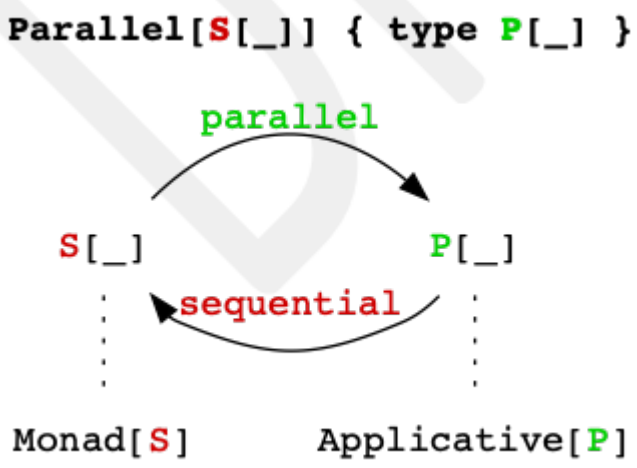

Figure 3. The `Parallel` typeclass encodes transformations between a sequential type `S` and a parallel type `P`.

Rewriting the translation between `IO` and `IO.Par` in terms of `Parallel`, we now have:

Example 8. Using `Parallel` *to translate between* `IO` *and* `IO.Par`

```
val ia: IO[A] = IO(???)
val ib: IO[B] = IO(???)

def f(a: A, b: B): C = ???

- val ipa: IO.Par[A] = IO.Par(ia)
- val ipb: IO.Par[B] = IO.Par(ib)
+ val ipa: IO.Par[A] = Parallel[IO].parallel(ia)
+ val ipb: IO.Par[B] = Parallel[IO].parallel(ib)

val ipc: IO.Par[C] = (ipa, ipb).mapN(f)

- val ic: IO[C] = IO.Par.unwrap(ipc)
+ val ic: IO[C] = Parallel[IO].sequential(ipc)
```

We can do better, though. Once a `Parallel` typeclass instance is defined, `par`-prefixed versions of functions become available *on the sequential type* that do this translation automatically, so you never see the underlying change of type:

Example 9. Replacing the explicit sequential → parallel → sequential transformation with the `parMapN` *method.*

```
val ia: IO[A] = IO(???)
val ib: IO[B] = IO(???)

def f(a: A, b: B): C = ???

- val ipa: IO.Par[A] = Parallel[IO].parallel(ia)
- val ipb: IO.Par[B] = Parallel[IO].parallel(ib)
-
- val ipc: IO.Par[C] = (ipa, ipb).mapN(f)
-
- val ic: IO[C] = Parallel[IO].sequential(ipc)
+ val ic: IO[C] = (ia, ib).parMapN(f) ①
```

① Notice the `par` prefix!

`parMapN` translates the arguments `ia` and `ib` to `IO.Par` values, composing them in parallel via `IO.Par` and `mapN`, and translates the results back to `IO`.

Look at how much code we've saved by abstracting over this common pattern!

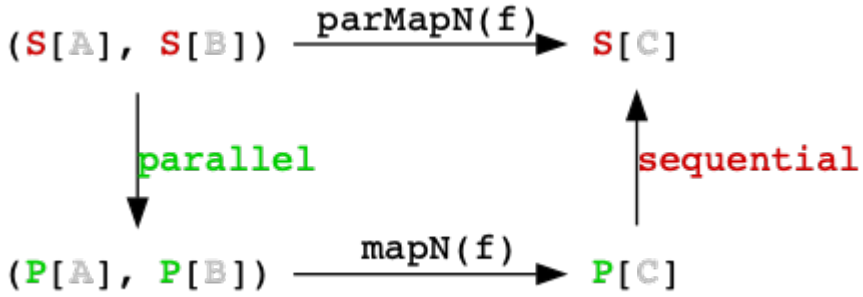

Figure 4. The `parMapN` *extension method is implemented as (1) translating the sequential effect types into parallel representations, (2) performing the alternative* `mapN`*, and (3) translating the parallel representation back to the sequential form.*

3.3. Inspecting parallelism

`parMapN` and other methods available from the `Parallel` type class usefully *abstract* parallelism, removing the details of how those computations actually occur. But when learning to use them, how do we get a feel for what is executing, and how?

The simplest solution would be to print something to the console during execution. We know that's a side effect, and we know what to do with side effects: wrap them in `IO`!

We've created a helper method, `debug`, to add to our code. Just add this import:

```
import com.innerproduct.ee.debug._
```

Now you can extend your effect values with the `debug` method:

Example 10. `parallel/DebugExample.scala`

```
package com.innerproduct.ee.parallel

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._

object DebugExample extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    seq.as(ExitCode.Success)

  val hello = IO("hello").debug ①
  val world = IO("world").debug ①

  val seq =
    (hello, world)
      .mapN((h, w) => s"$h $w")
      .debug ①
}
```

① Uses `debug` to add console output during execution.

At runtime, the `debug` method will print the name of the current thread, along with the value produced by the effect (as a string produced by invoking `toString`):

```
[ioapp-compute-0] hello       ①
[ioapp-compute-0] world       ①
[ioapp-compute-0] hello world ①
```

① Execution of the `seq` action using `mapN`. From the thread names, we see it runs entirely on the same thread.

The source for `debug` is very simple—it produces a new effect that prints the value of a given effect to the console, along with the name of the current thread:

Example 11. `debug.scala`

```
package com.innerproduct.ee

import cats.effect._

/** `import com.innerproduct.ee.debug._` to access
 * the `debug` extension methods. */
object debug {
  /** Extension methods for an effect of type `IO[A]`. */
  implicit class DebugHelper[A](ioa: IO[A]) {

    /** Print to the console the value of the effect
     * along with the thread it was computed on. */
    def debug: IO[A] =
      for {
        a <- ioa
        tn = Thread.currentThread.getName
        _ = println(s"[${Colorize.reversed(tn)}] $a") ①
      } yield a
  }
}
```

① We use another helper so that in a terminal the thread names will be given pretty colors to make them more visually distinct.

3.4. parMapN

`parMapN` is the parallel version of the applicative `mapN` method. It lets us combine multiple effects into one, *in parallel*, by specifying how to combine the outputs of the effects:

```
val ia: IO[A] = IO(???)
val ib: IO[B] = IO(???)

def f(a: A, b: B): C = ???

val ic: IO[C] = (ia, ib).parMapN(f)
```

`mapN` and `parMapN` act on tuples of any arity, so we can combine any number of effects together in a consistent way. For example:

```
        (ia, ib).parMapN((a, b)       => ???) ①
    (ia, ib, ic).parMapN((a, b, c)    => ???) ②
(ia, ib, ic, id).parMapN((a, b, c, d) => ???) ③
```

① *Two* effects → one effect.

② *Three* effects → one effect.

③ *Four* effects → one effect.

Let's make an example application that uses `parMapN`, along with our `debug` to let us see what's going on.

Example 12. `parallel/ParMapN.scala`

```
package com.innerproduct.ee.parallel

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._

object ParMapN extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    par.as(ExitCode.Success)

  val hello = IO("hello").debug ①
  val world = IO("world").debug ①

  val par =
    (hello, world)
      .parMapN((h, w) => s"$h $w") ②
      .debug ③
}
```

① We `debug` each `IO` value that will be executed (in parallel).

② We're using `parMapN` instead of `mapN` in the previous example.

③ We also `debug` the composed `IO` value. What do you think will be printed? What thread do you think it will run on?

Running the `ParMapN` program produces:

```
[ioapp-compute-1] world       ①
[ioapp-compute-0] hello       ①
[ioapp-compute-0] hello world ①
```

① Execution of the `tasks` action using `parMapN`. Notice the different threads that are

used!

The execution order of parallel tasks is non-deterministic, so you may see `hello` and `world` be printed in a different order when you run the program.

3.4.1. `parMapN` behavior in the presence of errors

Here's a program which prints the output of three `parMapN` composed effects, each of which represents a permutation of success and failure effects. What happens if one (or more) of the input effects has an error? What value is returned? Is it deterministic?

Example 13. `parallel/ParMapNErrors.scala`*: What happens if errors occur during* `parMapN`*?*

```
package com.innerproduct.ee.parallel

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._

object ParMapNErrors extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    e1.attempt.debug *> ①
      IO("---").debug *>
      e2.attempt.debug *>
      IO("---").debug *>
      e3.attempt.debug *>
      IO.pure(ExitCode.Success)

  val ok = IO("hi").debug
  val ko1 = IO.raiseError[String](new RuntimeException("oh!")).debug
  val ko2 = IO.raiseError[String](new RuntimeException("noes!")).debug

  val e1 = (ok, ko1).parMapN((_, _) => ())
  val e2 = (ko1, ok).parMapN((_, _) => ())
  val e3 = (ko1, ko2).parMapN((_, _) => ())
}
```

① Recall `attempt` transforms an `IO[A]` into an `IO[Either[Throwable, A]]`, ensuring the effect will always succeed (but with a `Left` value if it actually failed). We use `attempt` to ensure our error experiments don't stop the program.

Running `ParMapNErrors` outputs:

```
[ioapp-compute-0] hi ①
[ioapp-compute-1] Left(java.lang.RuntimeException: oh!) ①
[ioapp-compute-2] Left(java.lang.RuntimeException: oh!) ②
[ioapp-compute-4] Left(java.lang.RuntimeException: oh!) ③
```

① Output from `val e1 = (ok, ko1).parMapN((_, _) => ()).`

② Output from `val e2 = (ko1, ok).parMapN((_, _) => ())`

③ Output from `val e3 = (ko1, ko2).parMapN((_, _) => ())`

All three effects result in a `Left` (failure).

Let's first describe what must be true for all of these effects: the result of `parMapN` will fail if any—at least one—of the effects fail. And we see that in the output there are three `Left` values produced, corresponding to `e1`, `e2`, and `e3`. At the same time, we know that each sub-effect (`ok`, `ko1`, `ko2`) are running in parallel.

Given these conditions, for the `e1` effect we see the output from `ok`, which means that `ok` executed before `ko1`. We *don't* see the output of `ok` from `e2`, so we can assume that the `e2` `ko1` effect happened first.

Both the `e1` `ok` effect and the `e2` `ko1` effect are the first arguments to `parMapN`. Does this mean that the left-most arguments to `parMapN` will always execute first? Not necessarily! Consider if we delay the execution of `ko1` with a sleep:

```
+ import scala.concurrent.duration._

- val ko1 = IO.raiseError[String](new RuntimeException("oh!")).debug
+ val ko1 =
+   IO.sleep(1.second).as("ko1").debug *>
+   IO.raiseError[String](new RuntimeException("oh!")).debug
```

`ParMapNErrors` then outputs:

```
[ioapp-compute-0] hi                                    ①
[ioapp-compute-1] ko1                                   ①
[ioapp-compute-1] Left(java.lang.RuntimeException: oh!) ①
[ioapp-compute-2] hi                                    ②
[ioapp-compute-3] ko1                                   ②
[ioapp-compute-3] Left(java.lang.RuntimeException: oh!) ②
[ioapp-compute-4] Left(java.lang.RuntimeException: noes!) ③
```

① We've delayed `ko1`, so for `e1` we see the output of `ok` and `ko1` before `ko1` triggers the exception.

② For `e2`, even though `ko1` is the first argument to `parMapN` we see the same output as `e1`.

③ For `e2` we see the output of `ko2`, since `ko1` was delayed and thus executed after `ko2`.

What happens if there are failures during `parMapN`? The *first* failure to happen is used as the failure of the composed effect.

3.4.2. `parTupled`

The `parMapN((_, _) => ())` code looks a bit ugly. We're doing two things with that expression:

1. No matter what the results of the inputs effects are, we want to produce a `Unit`.
2. Because we don't care what the two results of the input effects are, we "name" them _ to ignore them.

To accomplish the first goal, we could use the `void` combinator, which is defined as `map(_ ⇒ ())`:

```
- val e1 = (ok, ko1).parMapN(???).map(_ => ())
+ val e1 = (ok, ko1).parMapN(???).void
```

But what can we put in place of the ??? The simplest function we can pass to `mapN` would be a function that doesn't do anything:

```
val e1 = (ok, ko1).parMapN((l, r) => (l, r)).void
```

`cats` provides a `(par-)mapN` function that doesn't do anything except tuple the inputs together, called `(par-)tupled`:

```
        (ia, ib).parTupled ①
    (ia, ib, ic).parTupled ②
(ia, ib, ic, id).parTupled ③
                       ... ④
```

① *Two* `IO` → one `IO` of a `Tuple2`: `(IO[A], IO[B]) ⇒ IO[(A, B)]`

② *Three* `IO` → one `IO` of a `Tuple3`: `(IO[A], IO[B], IO[C]) ⇒ IO[(A, B, C)]`

③ *Four* `IO` → one `IO` of a `Tuple4`: `(IO[A], IO[B], IO[C], IO[D]) ⇒ IO[(A, B, C, D)]`

④ And so on.

So our error-handling examples above could be written:

```
- val e1 = (ok, ko1).parMapN((l, r) => (l, r)).void
+ val e1 = (ok, ko1).parTupled.void
```

3.5. parTraverse

parTraverse is the parallel version of traverse; both have the type signature:

```
F[A] => (A => G[B]) => G[F[B]]
```

For example, if F is List and G is IO, then (par)traverse would be a function from a List[A] to an IO[List[B]] when given a function A ⇒ IO[B].

```
List[A] => (A => IO[B]) => IO[List[B]]
```

The most common use case of (par)traverse is when you have a collection of work to be done, and a function which handles *one* unit of work. Then you get a collection of results combined into *one* effect:

```
val work: List[WorkUnit] = ???
def doWork(workUnit: WorkUnit): IO[Result] = ??? ①

val results: IO[List[Result]] = work.parTraverse(doWork)
```

① Note that processing one unit of work is an **effect**, in this case, IO.

Let's use our debug combinator to better see the execution when using parTraverse.

Example 14. `parallel/ParTraverse.scala`*: What output to you expect to be printed?*

```
package com.innerproduct.ee.parallel

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._

object ParTraverse extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    tasks
      .parTraverse(task) ①
      .debug ②
      .as(ExitCode.Success)

  val numTasks = 100
  val tasks: List[Int] = List.range(0, numTasks)

  def task(id: Int): IO[Int] = IO(id).debug ②
}
```

① We `parTraverse` over the `tasks`: each `Int` of `tasks` is transformed into an `IO[Int]` via the `task` method, and they are executed in parallel.

② We use the `debug` combinator on each task effect, and the final result effect.

Running the `ParTraverse` program produces:

```
[ioapp-compute-7] 7
[ioapp-compute-0] 0
[ioapp-compute-6] 6
[ioapp-compute-1] 1
[ioapp-compute-2] 2
[ioapp-compute-5] 5
[ioapp-compute-4] 4
[ioapp-compute-3] 3
[ioapp-compute-7] 8
[ioapp-compute-3] 13
...
[ioapp-compute-4] List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
97, 98, 99)
```

If all results are computed in parallel, how is the `List[B]` of results created by the returned `IO[List[B]]`?

To produce a result of type `IO[List[B]]` must mean the returned `IO` must have

collected all of the results—the `List[B]`—even though each `B` was computed independently.

It's necessary to wait until all the elements have have been traversed, but the returned `List[B]` can be incrementally built by waiting for the first result to be computed, then append the second result when it is computed, and so on.

That being said, `parTraverse` is actually written in terms of `traverse`, where it transforms every `IO` into `IO.Par`. Since `traverse` only requires the effect to have an `Applicative` instance, the `Applicative[IO.Par]` is where the parallelism "happens".

3.5.1. Another view of `parTraverse`

You can also think of `(par)traverse` as a variation of `(par)mapN` where results are collected, but where every input effect has the same output type:

```
def f(i: Int): IO[Int] = IO(i)

           (f(1), f(2)).parMapN((a, b)       => List(a, b))       // IO[List[Int]] ①
      (f(1), f(2), f(3)).parMapN((a, b, c)    => List(a, b, c))    // IO[List[Int]] ②
(f(1), f(2), f(3), f(4)).parMapN((a, b, c, d) => List(a, b, c, d)) // IO[List[Int]] ③
       List(1, 2, 3, 4).parTraverse(f)                             // IO[List[Int]] ④
```

① We compute `f(1)`, `f(2)`, and collect the results into a List.

② We compute `f(1)`, `f(2)`, `f(3)`, and collect the results into a List.

③ We compute `f(1)`, `f(2)`, `f(3)`, `f(4)`, and collect the results into a List.

④ `List(1, 2, 3, 4).parTraverse(f)` is the same as `(f(1), f(2), f(3), f(4)).parMapN(⋯)`.

Notice the return type for all of these expressions is the same: `IO[List[Int]]`.

3.6. `parSequence`

`(par)sequence` turns a nested structure "inside-out":

```
F[G[A]] => G[F[A]]
```

For example, if you have a `List` of `IO` effects, `parSequence` will, in parallel, transform it into *one* `IO` effect that produces a `List` of outputs:

```
List[IO[A]] => IO[List[A]]
```

Let's see parSequence in action:

Example 15. parallel/ParSequence.scala*: What output to you expect to be printed?*

```
package com.innerproduct.ee.parallel

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._

object ParSequence extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    tasks.parSequence ①
      .debug ②
      .as(ExitCode.Success)

  val numTasks = 100
  val tasks: List[IO[Int]] = List.tabulate(numTasks)(task)

  def task(id: Int): IO[Int] = IO(id).debug ②
}
```

① We parSequence over the tasks: each IO[Int] of tasks is executed in parallel.

② We use the debug combinator on each task effect, and the final result effect.

Running the ParSequence program produces:

```
[ioapp-compute-2] 2
[ioapp-compute-4] 4
[ioapp-compute-1] 1
[ioapp-compute-7] 7
[ioapp-compute-3] 3
[ioapp-compute-0] 0
[ioapp-compute-2] 8
[ioapp-compute-6] 6
[ioapp-compute-1] 12
...
[ioapp-compute-6] List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
97, 98, 99)
```

Note that sequence and traverse are mutually definable: x.sequence is x.traverse(identity), and x.traverse(f) is x.map(f).sequence.

3.7. Summary

1. `IO` does not support parallel operations itself, because it is a `Monad`.
2. The `Parallel` typeclass specifies the translation between a pair of effect types: one that is a `Monad` and the other that is "only" an `Applicative`.
3. `Parallel[IO]` connects the `IO` effect to its parallel counterpart, `IO.Par`.
4. Parallel `IO` composition requires the ability to shift computations to other threads within the current `ExecutionContext`. This is how parallelism is "implemented".
5. `parMapN`, `parTraverse`, `parSequence` are the parallel versions of (the sequential) `mapN`, `traverse`, and `sequence`. Errors are managed in a fail-fast manner.

[10] It's also a consequence of what's termed *typeclass coherence*. Typeclass coherence says there should only be *one* instance of a typeclass per type, so there can only be one `Monad` instance for `IO`. At the same time, since every `Monad` is also an `Applicative`, then by coherence the unique `Applicative` for `IO` must be that same `Monad` instance for `IO`. This also implies that the use of `Applicative` methods, like `mapN`, should be equivalent to the use of `Monad` methods like `flatMap`; that is, if you *really* have a `Monad`, rewriting a monadic expression like `fa.flatMap(_ => fb)` to `(fa, fb).mapN((_, b) => b)` produces the "same" program. You shouldn't get different behavior by "forgetting" a type is a `Monad`.

Chapter 4. Concurrent control

So far we've been working with rather opaque effects: we can describe them and eventually run them to produce a value (or an error). But we don't yet have any way to control a *running* computation.

Example 16. Without concurrent control, we can only describe and (eventually) run effects.

```
val i1: IO[A] = ??? ①
val i2: IO[B] = ??? ①
val i3: IO[C] = doSomething(i1, i2) ①

val c: C = i3.unsafeRunSync() ②
```

① These effects haven't started yet. We've only described *what* they compute.

② We get the *result* of the computation when it is complete. We don't have access to *how* it is computed, so we can't affect (control) it.

Because a computation may be running, to control it means we will be acting *concurrently* with it. In this chapter we'll discuss how to *fork* and *join* a concurrent effect, *cancel* a concurrently running effect, and how to *race* multiple effects concurrently.

Concurrency vs. parallelism

Although they are often conflated, *concurrent* and *parallel* are disparate concepts:

concurrent Computations are concurrent when their execution *lifetimes* overlap.

parallel Computations are parallel when their executions occur at the same *instant* in time.

That is to say, concurrency is about the looking at the *structure* of the computations and how their lifetimes align, whereas parallelism is more about the operational utilization of *resources* during the execution.

For example, with two threads you could run two computations in parallel (and concurrently!). But with one thread you could also run two computations concurrently: if you can "pause" one and switch, using the same thread, to the other, and vice-versa, they would execute concurrently.

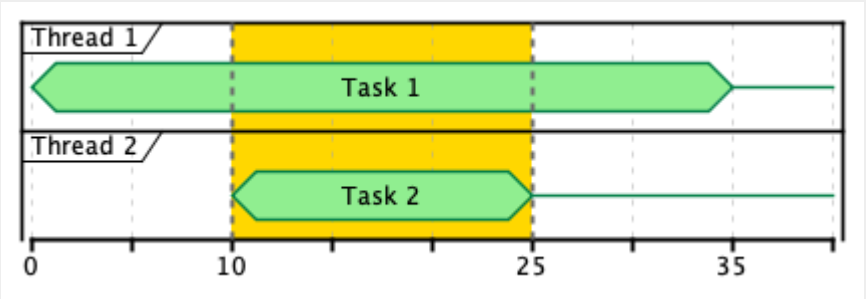

Figure 5. Two threads allow computations to execute in parallel during the highlighted period. They are also executing concurrently.

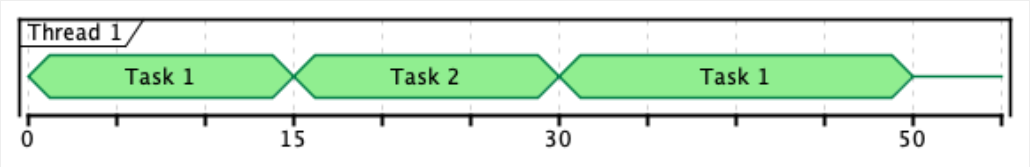

Figure 6. Two computations could execute concurrently with only one thread, if a computation can be interrupted and resumed.

Concurrency emphasizes the *non-deterministic* aspects of computation: we can't tell when anything happens, only that their lifetimes overlap. Whereas parallelism *requires* determinism: no matter how many resources you have, you *must* produce the same answer.

4.1. Decomposing the behavior of `parMapN`

To demonstrate forking, joining, and cancelation of concurrent effects, we'll write our own version of `parMapN`, which involves each of them.

```
def myParMapN[A, B, C](ia: IO[A], ib: IO[B])(f: (A, B) => C): IO[C] =
  ???
```

`myParMapN`, just like `parMapN`, needs to:

- ▢ start both the `ia` and `ib` computations so they run concurrently ("fork" them);
- ▢ wait for each result;
- ▢ cancel the "other" effect if `ia` or `ib` fails; and
- ▢ finally combine the results with the `f` function.

(We're only writing the two-argument variation of `parMapN`, ignoring the other arities.)

It's important to note that in order to "wait" and "cancel", we'll need *something* to "wait" and "cancel" on, a kind of handle to the "started" computation. In Cats Effect that concept is a *fiber*.

4.2. Gaining control with `Fiber`

When we write an expression like

```
for {
  result <- effect
  ...
```

the value `result` only exists once it is produced by the effect. We're essentially waiting until the result is available to continue the computation. Instead of waiting for the result, we could instead *fork* an effect: the effect will be started, but we aren't interested in waiting for its completion.[11] The result of forking, however, will be a value that lets us manage the forked effect: a *fiber*.[12]

In Cats Effect, to fork an effect we'll use the `start` method. Let's make a simple program and examine its behavior:

Example 17. Forking an effect with `start`. *Code available at* `control/Start.scala`.

```
package com.innerproduct.ee.control

import cats.effect._
import com.innerproduct.ee.debug._

object Start extends IOApp {

  def run(args: List[String]): IO[ExitCode] =
    for {
      _ <- task.start ①
      _ <- IO("task was started").debug ②
    } yield ExitCode.Success

  val task: IO[String] =
    IO("task").debug
}
```

① We `start` the effect to fork its execution from the current effect.

② Immediately after we `start` the `task` we print something to the console.

Running `Start` we see the following output:

```
[ioapp-compute-1] task ①
[ioapp-compute-0] task was started
```

① Note the effect of `task` runs on the `ioapp-compute-1` thread, which is different than the following effect!

When you `start` an effect its execution is "forked": it is shifted to a different thread.

Here's the (simplified) signature of `start`:[13]

```
def start: IO[Fiber[IO, A]]
```

The return type is interesting—it returns a `Fiber`, a data type which lets us act on the `start`-ed effect. But why does `start` return the `Fiber` inside an `IO`?

It returns a `Fiber` inside an `IO` because if it instead produced, directly, a `Fiber`, that would mean our original `IO` is running *right now*, but in reality it isn't. The source `IO` only executes when we explicitly run it, so we need to delay access to this fiber—by wrapping it in an effect—until the source `IO` is executed.

Now that we've demonstrated forking a `Fiber`, we feel the need to offer a warning: a `Fiber` is a very "low-level" mechanism for concurrent control. While it's absolutely necessary for implementing the concurrency and parallelism of Cats Effect, as a developer you can often better achieve your goals by using higher-level abstractions and operations.

4.2.1. Continuing `myParMapN`: forking effects

We can use `start` to fork a concurrent effect, so let's use it for our `myParMapN` function:

```
def myParMapN[A, B, C](ia: IO[A], ib: IO[B])(f: (A, B) => C): IO[C] =
  for {
    fiberA <- ia.start ①
    fiberB <- ib.start ①
  } yield ??? ②
```

① We `start` each effect to run them concurrently.

② We don't yet know how to gather their results or possibly cancel them.

Here's our progress for the requirements:

- ☑ start both the `ia` and `ib` computations so they run concurrently ("fork" them);
- ☐ wait for each result;
- ☐ cancel the "other" effect if `ia` or `ib` fails; and
- ☐ finally combine the results with the `f` function.

4.2.2. Joining a running `Fiber`

When we call `start` on an `IO[A]` value we receive a `Fiber[IO, A]` value. It lets us talk *about* the execution of an `IO[A]` computation.

What can we do with a `Fiber`? The first thing we can do is to `join` it, which will return the result of the forked `IO` effect. We're giving up the control the fiber gave us, and subsequently we can only talk about the eventual result of the previously-forked value.

```
val joined: IO[String] =
  for {
    fiber <- IO("task").start
    s <- fiber.join
  } yield s
```

What happens if we `join` the `Fiber` that we just `start`-ed? What executes on which

thread?

Example 18. `control/JoinAfterStart.scala`

```
package com.innerproduct.ee.control

import cats.effect._
import com.innerproduct.ee.debug._
import scala.concurrent.duration._

object JoinAfterStart extends IOApp {

  def run(args: List[String]): IO[ExitCode] =
    for {
      fiber <- task.start
      _ <- IO("pre-join").debug
      _ <- fiber.join.debug ②
      _ <- IO("post-join").debug
    } yield ExitCode.Success

  val task: IO[String] =
    IO.sleep(2.seconds) *> IO("task").debug ①
}
```

① We introduce a delay and `debug` for the task to help us distinguish between the concurrent control of the task (use of the `Fiber`) and the task itself.

② After we print our pre-join message, we invoke `join`.

As a reminder, the `*>` extension method is equivalent to using `mapN` with two effects but only the second effect's value is produced; for example, `first *> second` is equivalent to `(first, second).mapN((_, b) ⇒ b)`.

Running `JoinAfterStart` outputs:

```
[ioapp-compute-0] pre-join
[ioapp-compute-1] task ①
[ioapp-compute-1] task
[ioapp-compute-1] post-join
```

① Notice that `task` is on a different thread than the "pre-join" output.

We also see `task` printed twice, once for the `IO("task").debug` and once for the `fiber.join.debug`.

When we `join` a `Fiber`, execution continues on the thread the `Fiber` was running on

(in this case, `ioapp-compute-1`).

4.2.3. Continuing `myParMapN`: joining forked effects

Now that we know we can await the results of a concurrent effect with `join`, we can update our `myParMapN` method. Since we need both results to invoke our transformation function `f`, it doesn't matter which order we `join` in. But we do need to `join` both forked tasks:

```
def myParMapN[A, B, C](ia: IO[A], ib: IO[B])(f: (A, B) => C): IO[C] =
  for {
    fiberA <- ia.start
    fiberB <- ib.start
    a <- fiberA.join ①
    b <- fiberB.join ②
  } yield f(a, b) ③
```

① Wait for the result of the forked `ia` via `fiberA`.

② Wait for the result of the forked `ib` via `fiberB`.

③ Once we have both, compute our desired value.

Here's our progress for the requirements:

- ☑ start both the `ia` and `ib` computations so they run concurrently ("fork" them);
- ☑ wait for each result;
- ☐ cancel the "other" effect if `ia` or `ib` fails; and
- ☑ finally combine the results with the `f` function.

We still need cancelation.

4.3. Canceling a running `Fiber`

The second thing we can do with a `Fiber` is to `cancel` it.

```
def cancel: cats.effect.CancelToken[IO]

type CancelToken[F[_]] = F[Unit] ①
```

① Canceling a `Fiber` is itself an effect. It produces a `Unit` value once the effect is canceled.

Why might we want to stop a running task? Usually it is because we've learned some information that tells us the computation isn't needed any longer. For example, we might `start` a fetch from a (relatively slow) datastore, but if the user

decides to cancel the overall operation, we should `cancel` the fetch to the underlying datastore.

Let's give a basic example of canceling a running `Fiber`:

Example 19. `control/Cancel.scala`

```
package com.innerproduct.ee.control

import cats.effect._
import cats.effect.implicits._
import com.innerproduct.ee.debug._

object Cancel extends IOApp {

  def run(args: List[String]): IO[ExitCode] =
    for {
      fiber <-
        task
          .onCancel(IO("i was cancelled").debug.void) ①
          .start
      _ <- IO("pre-cancel").debug
      _ <- fiber.cancel ②
      _ <- IO("canceled").debug
    } yield ExitCode.Success

  val task: IO[String] =
    IO("task").debug *>
      IO.never ③
}
```

① We add a `onCancel` callback to print to the console if the effect is cancelled. `onCancel` is an extension method to `IO` provided by `import cats.effect.implicits._`.

② We cancel the `Fiber` after the `start`.

③ `IO.never` is a built-in non-terminating effect. It has type `IO[Nothing]`, so since type `Nothing` is a type with no values, this effect can never complete. But it can be cancelled.

Running `Cancel` outputs:

```
ioapp-compute-0] pre-cancel
[ioapp-compute-1] task
[ioapp-compute-0] i was cancelled
[ioapp-compute-0] canceled
```

Note that `cancel` is idempotent. Invoking it more than once has the same effect as invoking it once—a canceled task will continue to be canceled.

However, if you `join` after you `cancel`, the `join` will never finish, because no result will ever be produced.

4.3.1. How does cancelation work?

Let's set up a situation where there's a long-lived effect running concurrently with an effect that produces an error. For the former we'll use the previously written "ticking clock":

```
val tickingClock: IO[Unit] =
  for {
    _ <- IO(System.currentTimeMillis).debug
    _ <- IO.sleep(1.second)
    _ <- tickingClock
  } yield ()
```

We'll run it concurrently with a failing effect using `parTupled`:

```
val ohNoes =
  IO.sleep(2.seconds) *> IO.raiseError(new RuntimeException("oh noes!")) ①

val together =
  (tickingClock, ohNoes).parTupled
```

① We raise an error after two seconds, to give the ticking clock a chance to print a few times to the console.

so that once the exception is raised, the `tickingClock` will be cancelled by some kind of "error handler" belonging to the `parMapN`-composed effect.

If we run the `together` effect we see:

```
[ioapp-compute-0] 1603147303459
[ioapp-compute-1] 1603147304469
java.lang.RuntimeException: oh noes!
    at com.innerproduct.ee.concurrent.CancelledClock$.<clinit>(CancelledClock.scala:16)
    at com.innerproduct.ee.concurrent.CancelledClock.main(CancelledClock.scala)
```

Our endlessly recursing `tickingClock` effect stops, and we didn't explicitly do anything. So how does cancelation work? And can our effects "know" if they've been canceled, and react to that information?

To define the behavior of cancelation, Cats Effect uses the concept of a *cancelation boundary*. As an effect executes, if a cancelation boundary—whatever that is—is encountered, then the cancelation status for the current effect is checked, and if that effect has been canceled then execution will stop.

From one perspective, cancelation is "automatic" because Cats Effect itself periodically inserts a cancelation boundary during effect execution.[14] Alternatively, one can "manually" insert a cancelation boundary with `IO.cancelBoundary`.[15]

4.3.2. Continuing `myParMapN`: cancelation-on-error behavior

If an error occurs during one of our effects, we need to cancel "the other" fiber. Let's use the `onError` combinator to handle each effect:

```
def myParMapN[A, B, C](ia: IO[A], ib: IO[B])(f: (A, B) => C): IO[C] =
  for {
    fiberA <- ia.start
    fiberB <- ib.start
    a <- fiberA.join.onError(_ => fiberB.cancel) ①
    b <- fiberB.join.onError(_ => fiberA.cancel) ②
  } yield f(a, b)
```

① If the computation of `fiberA` has an error, cancel `fiberB`.

② If the computation of `fiberB` has an error, cancel `fiberA`.

However, there is a critical bug here. Can you guess what it is?

The issue is that registering an `onError` handler is itself an effect, so in the code above the handler would only be registered if we couple it to the result of `fiberA.join`. But if we do that, then we *won't* be registering the `onError` handler with the result of `fiberB` until after `fiberA` has actually finished:

```
def myParMapN[A, B, C](ia: IO[A], ib: IO[B])(f: (A, B) => C): IO[C] =
  for {
    fiberA <- ia.start
    fiberB <- ib.start
    a <- fiberA.join.onError(_ => fiberB.cancel)
    b <- fiberB.join.onError(_ => fiberA.cancel) ①
  } yield f(a, b)
```

① The `onError` handler for `fiberB` won't be registered until `fiberA` completes, not

what we want!

We need to instead ensure that *both* `onError` handlers are registered. If only we could write

```
for {
  fa <- ia.start
  fb <- ib.start
  faj = fa.join.onError(_ => fb.cancel)
  fbj = fb.join.onError(_ => fa.cancel)
  c <- myParMapN(
    fa.join.onError(_ => fb.cancel),
    fb.join.onError(_ => fa.cancel))(f)
} yield c
```

but that would be using the method we are trying to write! (And it would incorrectly handle cancelation). If we tried something "clever" like

```
for {
  fa <- ia.start
  fb <- ib.start
  faj = fa.join.onError(_ => fb.cancel)
  fbj = fb.join.onError(_ => fa.cancel)
  registerA <- faj.start ①
  registerB <- fbj.start ①
  a <- registerA.join
  b <- registerB.join
  c = f(a, b)
} yield c
```

① Attempt to register both `onError` handlers by forking (again).

this too will not properly handle cancelation: if one of the effects is cancelled, then a *subsequent* `join` will never complete.

We're stuck: we need to avoid doing a `join` on a potentially cancelled effect, but here either effect could be cancelled first—we don't know which. The `Fiber` API isn't expressive enough to give us the information we need. To solve the problem, we need a different "primitive" operation: we'll instead *race* two effects, which will let us know which effect finishes first so that we can subsequently `join` the other effect.

4.4. Racing multiple effects

When we compose multiple effects concurrently with `parMapN`, we provide a

function to transform the gathered output of *every* concurrently executing effect. What if instead we were only interested in the effect that completed first, relating them temporally. We call this a *race*, and can have one using the `IO.race` combinator:

```
def race[A, B](lh: IO[A], rh: IO[B])(implicit cs: ContextShift[IO]): IO[Either[A, B]]
```

You can think of `race` in relation to `parTupled`—both run the effects concurrently, but `parTupled` gives you both results (the first *and* the second), whereas `race` gives you only one (the first *or* the second):

```
val ia: IO[A] = ???
val ib: IO[B] = ???

(ia, ib).parTupled // IO[(A, B)]       ①
IO.race(ia, ib)    // IO[Either[A, B]] ②
```

① The produced `(A, B)` is an `A` *and* a `B`.

② The produced `Either[A, B]` is either an `A` *or* a `B`.

One particularly useful kind of race is a timeout for an effect: we race the effect against a corresponding "sleep" effect. If the sleep finishes before the main effect, a timeout has occurred.

Example 20. `control/Timeout.scala`

```
package com.innerproduct.ee.control

import cats.effect._
import cats.effect.implicits._
import com.innerproduct.ee.debug._
import scala.concurrent.duration._

object Timeout extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    for {
      done <- IO.race(task, timeout) ①
      _ <- done match { ②
        case Left(_)  => IO("   task: won").debug ③
        case Right(_) => IO("timeout: won").debug ④
      }
    } yield ExitCode.Success

  val task: IO[Unit]    = annotatedSleep("   task", 100.millis) ⑥
  val timeout: IO[Unit] = annotatedSleep("timeout", 500.millis)

  def annotatedSleep(name: String, duration: FiniteDuration): IO[Unit] =
    (
      IO(s"$name: starting").debug *>
      IO.sleep(duration) *> ⑤
      IO(s"$name: done").debug
    ).onCancel(IO(s"$name: cancelled").debug.void).void
}
```

① `IO.race` races two effects, and returns the value of the first to finish. The loser of the race is cancelled.

② `done` is a value of type `Either[Unit, Unit]`. We pattern match on the `Either` to handle each case.

③ If it was a `Left`, then `task` finished first, and `timeout` was cancelled.

④ If it was a `Right`, then `timeout` finished first, and `task was cancelled`.

⑤ Here we produce a sleep effect for a given duration.

⑥ What happens if we change the duration to `1000.millis` here?

Running this program prints out:

```
[ioapp-compute-1]    task: starting
[ioapp-compute-2] timeout: starting
[ioapp-compute-3]    task: done
[ioapp-compute-3] timeout: cancelled
[ioapp-compute-3]    task: won
```

This pattern is so common there's a built-in combinator: `IO.timeout`. The example above could be rewritten as below, although we lose the ability to directly handle if the timeout occurred or not:

```
+ _ <- task.timeout(500.millis) ①
- done <- IO.race(task, timeout)
- _ <- done match {
-        case Left(_)  => IO("   task: won").debug
-        case Right(_) => IO("timeout: won").debug
-      }
```

① Raises a `java.util.concurrent.TimeoutException` if the effect takes longer than the timeout duration.

If you do want to act when a timeout occurs instead of only having the effect canceled, you could use the `IO.timeoutTo` method which lets you provide an alternative `IO` value to evaluate if the timeout expires.

4.4.1. Racing without automatic cancelation

`IO.race` is built upon a simpler combinator, `IO.racePair`, which doesn't provide cancelation of the "losing" effect. Instead you receive the "winning" value along with the `Fiber` of the race "loser", so you can decide what you want to do with it.

```
def racePair[A, B](lh: IO[A], rh: IO[B])(implicit cs: ContextShift[IO]):
    IO[Either[(A, Fiber[IO, B]), (Fiber[IO, A], B)]] ①
```

① If either of the effects has an error, the other is cancelled.

With `racePair`, we can complete our implementation of cancelation-on-error for `myParMapN`:

```
def myParMapN[A, B, C](ia: IO[A], ib: IO[B])(f: (A, B) => C): IO[C] =
  IO.racePair(ia, ib).flatMap {
    case Left((a, fb))  => (IO.pure(a), fb.join).mapN(f) ①
    case Right((fa, b)) => (fa.join, IO.pure(b)).mapN(f) ①
  }
```

① If no errors occur, we'll detect which finishes first, wrap the value using `IO.pure` (it's already computed, so we don't need to use `IO.delay`), and then `join` the other until completion. Finally we combine the `IO` values with our function `f`.

We're done with `myParMapN`:

- ☑ start both the `ia` and `ib` computations so they run concurrently ("fork" them);
- ☑ wait for each result;
- ☑ cancel the "other" effect if `ia` or `ib` fails; and
- ☑ finally combine the results with the `f` function.

If you feel a bit cheated relying on `racePair` to register the cancelation for us, that's alright, you're entitled to feeling that way. `Fiber` itself doesn't give us enough control to implement cancelation-on-error.

4.5. Summary

1. Concurrency allows us to control running computations.
2. A `Fiber` is our handle to this control. After we start a concurrent computation, we can cancel or join it (wait for completion).
3. Concurrently executing effects can be cancelled. Cancelled effects are expected to stop executing via implicit or explicit cancelation boundaries.
4. We can race two computations to know who finished first. Higher-order effects like timeouts can be constructed using races.

[11] The term *fork* is meant to evoke the point at which, say, a river branches into two separate flows. One branch is the newly started effect, while the other is the continuation of the previously executing one.

[12] The term *fiber* was chosen as similar, but distinct variation of the term *thread*. Fibers in Cats Effect, however, are logically separate from any threads used during execution.

[13] In Cats Effect 2, `start` takes an implicit `ContextShift` parameter which represents the thread pool where effects are eventually run. In Cats Effect 3, the `start` method returns type `IO[FiberIO[A]]`, where `FiberIO[A]` is an alias to the underlying `Fiber[IO, Throwable, A]` base type, which has three type parameters vs. two type parameters in Cats Effect 2.

[14] In Cats Effect 2, a cancelation boundary is inserted after every 512 `flatMap` calls. In Cats Effect 3, every `flatMap` is treated as cancelation boundary.

[15] `IO.cancelBoundary` is removed in Cats Effect 3, since `flatMap` itself is defined as a cancelation boundary.

Chapter 5. Shifting contexts

Parallelism makes use of a set of resources to execute effects. On the JVM, this is a thread pool: effects execute on the available threads simultaneously. Scala's main abstraction for using thread pools is the `scala.concurrent.ExecutionContext`, and Cats Effect builds on top of it to implement parallelism and concurrency.

In this chapter we'll explore how these contexts are used by our `IOApp` programs and how different kinds of work—blocking vs. non-blocking—can require different execution strategies.

5.1. How much parallelism can we get?

So far our parallel and concurrent code has used whatever threads our `IOApp` gives us. How much work can we really do with it? For example, if we try to run a lot of effects in parallel, how many *actually* run in parallel? Let's experiment:

Example 21. `contexts/Parallelism.scala`*: How many effects can run in parallel?*

```
package com.innerproduct.ee.contexts

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._

object Parallelism extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    for {
      _ <- IO(s"number of CPUs: $numCpus").debug
      _ <- tasks.debug
    } yield ExitCode.Success

  val numCpus = Runtime.getRuntime().availableProcessors() ①
  val tasks = List.range(0, numCpus * 2).parTraverse(task) ②
  def task(i: Int): IO[Int] = IO(i).debug ③
}
```

① We ask for the number of CPUs available so we can ensure we submit *more* than this number of tasks.

② We want to run a large number of tasks in parallel; let's try twice the number of CPUs available.

③ Each effect we execute is a "no-op"; it doesn't do anything.

What do we see when we run it? On my machine it prints:

```
[ioapp-compute-0] number of CPUs: 8
[ioapp-compute-1] 1
[ioapp-compute-7] 7
[ioapp-compute-5] 5
[ioapp-compute-4] 4
[ioapp-compute-2] 2
[ioapp-compute-3] 3
[ioapp-compute-5] 8
[ioapp-compute-3] 9
[ioapp-compute-5] 11
[ioapp-compute-6] 6
[ioapp-compute-5] 15
[ioapp-compute-0] 0
[ioapp-compute-4] 14
[ioapp-compute-3] 13
[ioapp-compute-7] 12
[ioapp-compute-1] 10
[ioapp-compute-1] List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

We can see from our `debug` information we are using eight threads: threads zero (`ioapp-compute-0`) through seven (`ioapp-compute-7`), which is the same as our `numCpus`. However we had more than `numCpus` tasks, so that must mean that our underlying thread pool has at most `numCpus` threads.

At the same time, we ran `parTraverse` with twice as many effects as CPUs. How does the system ensure all the effects are run?

The answer is that when we compose effects in parallel, during execution each effect is only *scheduled* to be executed, and a separate asynchronous process is responsible for executing the scheduled effects on an available thread. When a thread finishes its work, another effect is executed on it.

In Scala this exactly maps to an `ExecutionContext`, which encapsulates both a queue of scheduled tasks and a set of threads used to execute them. In Cats Effect, every IOApp has a default `ExecutionContext`, and on the JVM it is constructed as a *fixed* pool based on the number of available CPUs. In all of our `IOApp`-based examples we've been using this hidden thread pool.

5.2. The need for multiple contexts

We saw in the previous example that we can only execute at most `numCpus` effects in parallel. This makes sense since we only have a `numCpus` CPUs! But at the same time, our computers regularly do more than `numCpus` things at the same time. How can we reconcile these disparate ideas?

The solution on the JVM is threads. We can have many threads running, and their execution is multiplexed across the available cores available from the operating system. And we can pool those threads into logical groups with data types like `ExecutionContext`, where threads in one pool are isolated from those in another.

Everything can work fine in this kind of world if we are computing with pure values. Threads will compete to be run and whatever priorities and fairness algorithms will be applied to ensure we make progress. But if we start interacting with the external environment, like reading from a file or writing to the network, our threads can become *blocked*. Data may not be available yet, the network hasn't acknowledged receiving anything yet, and so on. We'll call the former kind of work "CPU-bound" and the latter—the blocking kind—"I/O-bound". (I/O refers to input/output, not the `cats.effect.IO` effect type.)

When a thread is blocked, the JVM suspends its execution so another thread can be executed by the operating system.[16] But at the same time, there can be limits to the number of possible threads, usually as part of configuring the underlying thread pool. What do we do if our pool has at most `n` threads, but all those threads are blocked? If that happens, we can't use any available cores to do CPU-bound work.

To ensure our programs make progress—ensuring work proceeds when I/O-bound work is blocked—we'll isolate the CPU-bound work from any I/O-bound tasks by having separate pools. The Cats Effect library supports this pattern by encouraging separate contexts:

- CPU-bound work will be scheduled on a fixed-size thread pool, where the number of threads is the number of cores available to the JVM. All things being equal, you can't compute more than `<number of CPUs>` things at a time, so don't try to do more.
- I/O-bound work will be scheduled on an *unbounded* thread pool so that blocked threads merely take up memory instead of stopping the progress of other tasks.

In an `IOApp` on the JVM the default `ExecutionContext` is configured for CPU-bound work.[17] The next section will answer the question: what context do we use for I/O-bound work?

Figure 7. Thread pool best-practices. Image by @impurepics.[18]

5.3. Contexts for I/O-bound actions

We could instantiate our own `ExecutionContext` to use for blocking I/O effects, configuring it to use an unbounded thread pool. But it would be somewhat difficult to properly use since it has the same type as any other `ExecutionContext` value, like the default context provided for CPU-bound work by `IOApp`. We could easily pass the wrong context to a method. If instead we had a context with a different type we couldn't make that mistake. Luckily Cats Effect provides the exact solution: `Blocker`.

Note: `Blocker` *only applies to Cats Effect 2. Managing blocking effects with Cats Effect 3 is discussed below.*

`Blocker` is a small wrapper around an `ExecutionContext`. Let's demonstrate creating a `Blocker` and using it to execute effects in the blocking context.

Example 22. `contexts/Blocking.scala`

```
package com.innerproduct.ee.contexts

import cats.effect._
import com.innerproduct.ee.debug._

object Blocking extends IOApp {

  def run(args: List[String]): IO[ExitCode] =
    Blocker[IO].use { blocker => ①
      withBlocker(blocker).as(ExitCode.Success)
    }

  def withBlocker(blocker: Blocker): IO[Unit] =
    for {
      _ <- IO("on default").debug
      _ <- blocker.blockOn(IO("on blocker").debug) ②
      _ <- IO("where am I?").debug ③
    } yield ()
}
```

① We can't directly instantiate a `Blocker`, but we can use `Blocker.apply[IO]` to create a `Resource[IO, Blocker]` that manages the underlying thread pool used for blocking computations.

To access the `Blocker` itself, we `use` the `Resource`, passing it a function consumes the `Blocker` and produces an effect. We'll discuss `Resource` in more depth in Chapter 7, *Managing resources*.

② To execute our effect on the blocking context, we provide it to the `blockOn` method of the `Blocker`.

Note that we attach the `debug` to the effect we want running in the `Blocker`! If we moved `debug` onto the `blockOn` call we'd see `on blocker` printed on the default `ContextShift` (`ioapp-compute-*`).

③ Subsequent effects execute on the original context, not the blocking one.

Running `Blocking` outputs:

```
[ioapp-compute-0] on default
[cats-effect-blocker-0] on blocker
[ioapp-compute-1] where am I?
```

We used the `blockOn` method of `Blocker` to declare an existent effect should run on

the blocking context. However, f we wanted to create a blocking effect directly, in one step, we could use the `delay` method of `Blocker`, analogous to the `IO.delay` method:

```
def blockingDebug[A](blocker: Blocker, a: => A): IO[A] =
  blocker.delay {
    val value = a
    println(s"[${Thread.currentThread.getName}] $value")
    value
  }
```

5.3.1. Declaring blocking effects in Cats Effect 3

Instead of using a `Blocker` backed by a special `ExecutionContext`, Cats Effect 3 gives us a dedicated effect constructor to declare an effect as blocking as early as possible:

```
val withBlocker: IO[Unit] =
  for {
    _ <- IO("on default").debug
    _ <- IO.blocking("on blocker").debug ①
  } yield ()
```

① We directly declare a blocking effect with `IO.blocking`. But be careful, the `debug` call here will run on the default context, not the blocking one, because blocking effects always shift back to the previous context.

5.4. How do you know something is blocking?

We now have a separate strategy for executing blocking effects using `Blocker`. But how do we know what we're doing is blocking or not? Daniel Spiewak, long-time Scala contributor and a maintainer of Cats Effect, offers us a heuristic:[19]

> if something doesn't have a callback API, then you know it's blocking
>
> — Daniel Spiewak, *Cats Effect gitter.im chatroom*

The idea being: a callback API allows the API to return immediately so the caller is not blocked while the API is computing the result; therefore if there *isn't* such a callback API, then the method is probably blocking. Methods that return values that themselves have a callback API, such as `scala.concurrent.Future` or `IO`, would imply those methods are not blocking.

Exercise 3: Collect some blocking APIs

Collect some examples of blocking methods. You could look in the Scala or Java standard library, or from one of your favorite libraries.

How do you know they block?

5.5. Finer-grained control of contexts

Cats Effect encourages a coarse (but useful!) distinction for executing effects: they are either CPU-bound or I/O bound, and are assigned by the programmer to either the default or blocking `ExecutionContext`, respectively. But there are two other scenarios that may occur that involve the relationship between an effect and its execution context: long-running effects, and effects that need to be executed in neither the default nor blocking context.

For the first scenario, let's re-examine a long-running effect, the ticking clock, implemented as a recursive function:

```
val tickingClock: IO[Unit] =
  for {
    _ <- IO(System.currentTimeMillis).debug
    _ <- IO.sleep(1.second)
    _ <- tickingClock
  } yield ()
```

Does this effect execute on one thread, forever? If it did, that would be bad, because `tickingClock` isn't really doing anything other than sleeping, and so to hoard the current thread for such an effect would make one less thread available for *other* effects to execute on, reducing the amount of work our applications can perform.

To ensure a recursive loop doesn't steal a thread and never give it back, we'd like to be able to declare, as an effect itself, "reschedule the remainder of the computation". Not only would this resume the computation on (potentially) another thread when the resumption is executed by the context, but it then allows other scheduled effects to re-use the previous thread. In other words, the current effect is "suspended" and sent "to the back of the line", which prevents other effects from being "starved" of a thread.

In Cats Effect, this notion of "reschedule the remainder of the computation" is an instance of a larger concept, an *asynchronous boundary*. From the point of view of the composed effect, the boundary marks where the runtime could reschedule the

computation to resume on another thread. We can produce an asynchronous boundary with the `IO.shift` method:

Example 23. Shifting multiple times with the same context. Code available at `contexts/Shifting.scala`.

```
package com.innerproduct.ee.contexts

import cats.effect._
import com.innerproduct.ee.debug._

object Shifting extends IOApp {

  def run(args: List[String]): IO[ExitCode] =
    for {
      _ <- IO("one").debug
      _ <- IO.shift
      _ <- IO("two").debug
      _ <- IO.shift
      _ <- IO("three").debug
    } yield ExitCode.Success
}
```

Running this example outputs:

```
[ioapp-compute-0] one
[ioapp-compute-1] two
[ioapp-compute-2] three
```

We can see that after every `shift`, the next effect will run in a different thread in the context.

Returning to our ticking clock, you may recall that when we ran it, multiple threads are used, even though we didn't add an asynchronous boundary with `IO.shift`. What's going on?

Example 24. Our ticking clock uses multiple threads!?

```
[ioapp-compute-0] 1607561985119
[ioapp-compute-1] 1607561986128
[ioapp-compute-2] 1607561987130
```

Even though we can't see it, there are asynchronous boundaries composed with

our ticking clock. They are introduced by the `IO.sleep` effect, and if you think about it, this makes sense, since if we actually blocked the current thread for the duration of the sleep, we'd be preventing that thread from being used by other effects.

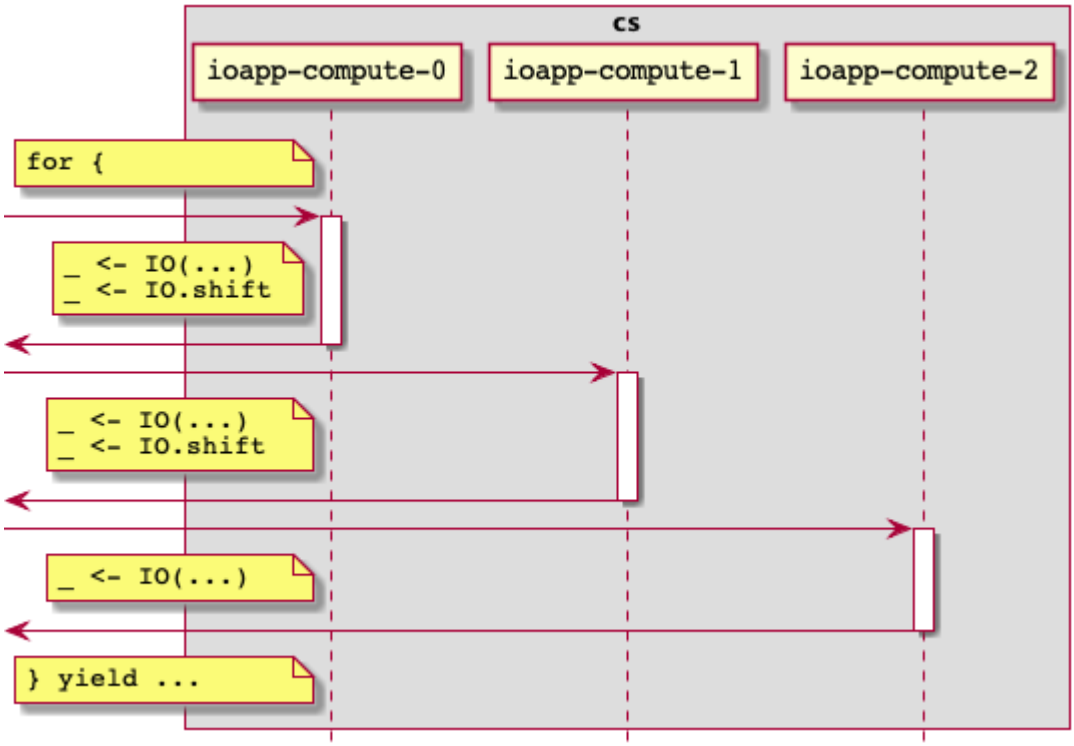

Figure 8. Shifting computations between threads in a context.

5.5.1. Shifting with multiple contexts

We've made long-running effects more fair with respect to other concurrently executing effects. What about effects we want to run neither on the default context, nor on a blocking one? (You might need this, for example, when integrating with a library that manages its own thread pools.) Luckily, we can expand the notion of an asynchronous boundary so that we can specify a *particular* context to resume our computation on, rather than the "current" one. Concretely, `IO.shift` can take an optional `ExecutionContext`:

Example 25. Shifting computation between contexts. Code available at `contexts/ShiftingMultiple.scala`.

```
package com.innerproduct.ee.contexts

import cats.effect._
import com.innerproduct.ee.debug._
import java.util.concurrent.Executors
import scala.concurrent.ExecutionContext

object ShiftingMultiple extends IOApp {

  def run(args: List[String]): IO[ExitCode] =
    (ec("1"), ec("2")) match { ①
      case (ec1, ec2) =>
        for {
          _ <- IO("one").debug ②
          _ <- IO.shift(ec1) ③
          _ <- IO("two").debug ③
          _ <- IO.shift(ec2) ④
          _ <- IO("three").debug ④
        } yield ExitCode.Success
    }

  def ec(name: String): ExecutionContext = ⑤
    ExecutionContext.fromExecutor(Executors.newSingleThreadExecutor { r =>
      val t = new Thread(r, s"pool-$name-thread-1")
      t.setDaemon(true) ⑥
      t
    })
}
```

① We construct two new `ExecutionContext` values to use.

② By default, our computations will execute on the `ExecutionContext` of our `IOApp`.

③ We `shift` onto another `ExecutionContext` for the next effect.

④ Next we `shift` onto the second `ExecutionContext`.

⑤ Some boilerplate to create a new single-threaded `ExecutionContext`.

⑥ We need daemon threads so the JVM shuts down correctly. We'll address this differently in Chapter 7, *Managing resources*.

Running this example outputs:

```
[ioapp-compute-0] one
[pool-1-thread-1] two
[pool-2-thread-1] three
```

During the entire program, the execution sequence moves between threads belonging to the various `ExecutionContext` values:

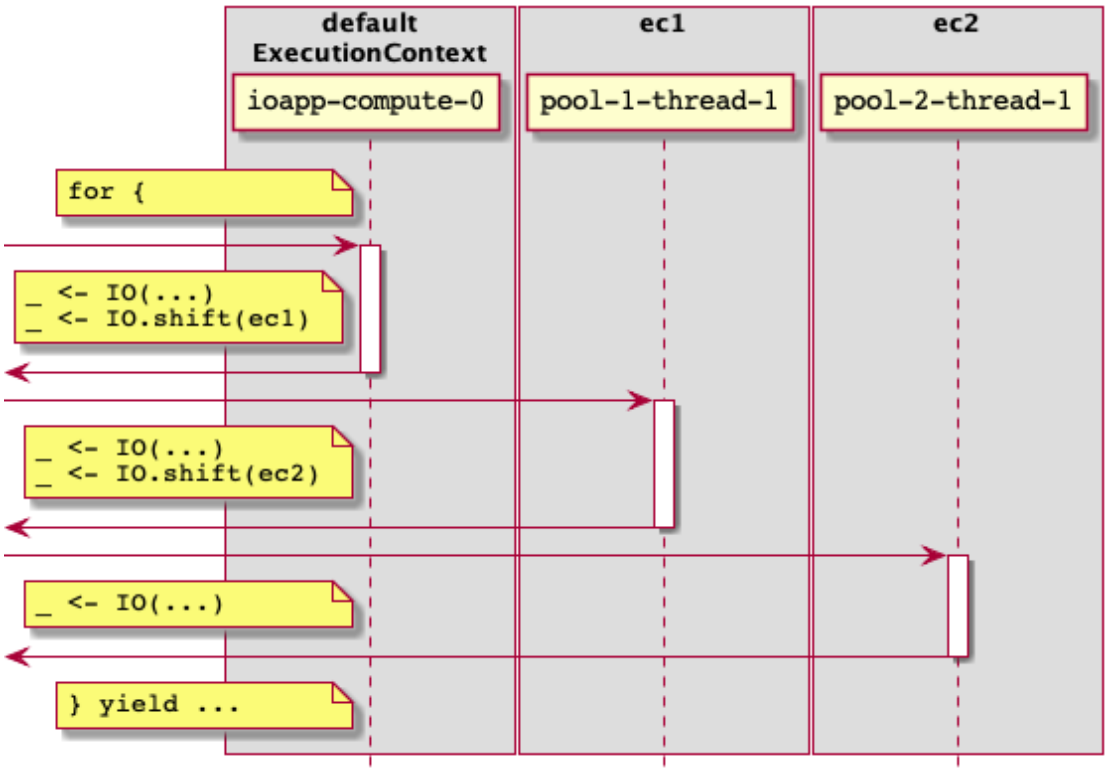

Figure 9. Shifting computations between multiple contexts.

5.6. Example: contexts for database access in Doobie

Doobie[20] is “a pure functional JDBC layer for Scala and Cats” that not only uses Cats Effect, but exemplifies the use of multiple contexts to handle different types of effects. These contexts are used by the `Transactor` data type:[21]

> Most doobie programs are values of type ConnectionIO[A] or Stream[ConnnectionIO, A] that describe computations requiring a database connection. By providing a means of acquiring a JDBC connection we can transform these programs into computations that can actually be executed. The most common way of performing this transformation is via a Transactor.
>
> — Managing Connections, Book of Doobie

A `Transactor` uses the multiple contexts to enforce certain constraints when communicating with a database over JDBC:

1. Acquiring a database connection should use a separate thread pool from other kinds of effects, since they block. However, if the underlying JDBC thread pool itself is fixed, instead of using a standard blocking context a separate fixed thread pool should be used, since using more threads than the underlying pool would only create additional blocked threads.
2. JDBC operations themselves block, so the usual blocking context should be used.
3. Any non-blocking pre- or post-query effects should be run on the default context.

We can now imagine what Doobie does in various places to ensure these constraints: inserting asynchronous barriers to shift the effect onto the correct thread pool for the situation.

5.7. Summary

1. Threads abstract over what is concurrently executing atop the available set of processors, so we can have many more threads than CPUs. A `scala.concurrent.ExecutionContext` represents a scheduling queue along with a set of threads used for computation.
2. Asynchronous boundaries help to ensure applications make progress in the presence of long-running effects by rescheduling the remainder of the effect. At the same time we can specify a computation to resume on a different context in order to isolate various workloads from one another.
3. `IOApp` provides a default `ExecutionContext` with a fixed number—the number of CPUs on the machine—of threads. This is meant for CPU-bound (non-blocking) work.
4. I/O-bound work, which is usually slower than CPU-bound work because it will

block the thread it uses, should run in a pool separate from CPU-bound work. Blocking I/O-bound work should be run in an *unbounded* thread pool. Cats Effect provides the `Blocker` interface to declare effects that block.

[16] This is a terribly simplified explanation. The Java Language Specification, Chapter 17: Threads and Locks is one place to learn more about the underlying model.

[17] The default `ExecutionContext` in an `IOApp` uses *daemon threads* so that if the top-level effect (specified by the `run` method) completes, any concurrently executing effects do not prevent the application from exiting.

[18] https://typelevel.org/cats-effect/docs/2.x/concurrency/basics#choosing-thread-pool

[19] https://gitter.im/typelevel/cats-effect?at=5e3b38cd6f9d3d34982283d4

[20] https://tpolecat.github.io/doobie

[21] https://tpolecat.github.io/doobie/docs/14-Managing-Connections.html#about-transactors

Chapter 6. Integrating asynchrony

While we hope to write programs that use safer datatypes like `cats.effect.IO`, our codebases can't be migrated overnight. We already use various built-in types like `scala.concurrent.Future`, along with other libraries to write parallel and concurrent code. How can we wrap them to instead produce `IO` values?

To answer this we'll discuss Cats Effect `IO.async` method, which uses the general pattern of continuation passing to integrate any kind of asynchronous processing interface.

6.1. Asynchronous callbacks

> You don't talk to me, I talk to you!
>
> — Stella, age 2

How can we integrate with any possible asynchronous interface? For that we're going to use the `IO.async` method to construct an `IO` value from a callback-based API. Remember—an API that provides callbacks implies that computation is happening asynchronously. After you provide a callback, you can do other work, and the callback will typically be executed on some other thread once the computation completes.

```
def async[A](k: (Either[Throwable, A] => Unit) => Unit): IO[A]
```

`async` is a higher-order function, in this case a function that takes another function as an argument. A perhaps more familiar example of a higher-order function is `map`, as in `List(1, 2, 3).map(_ + 1)`. Why do we use this technique? We pass functions to other functions when we need *them* to call it, not us: `map` invokes—for us—the `_ + 1` function for each element of the list.

The `async` signature is quite complex and a bit difficult to read. If we create a type alias for part of the signature, it becomes a bit easier to understand:

```
- def async[A](k: (Either[Throwable, A] => Unit) => Unit): IO[A]
+ type Callback[A] = Either[Throwable, A] => Unit ①
+
+ def async[A](k: CallBack[A] => Unit): IO[A]
```

① A callback is a function that *receives* the result of a computation. In this case, the result is either an error (`Throwable`) or a successful value of type `A`.

Breaking down its signature, we provide `async` a function `k`, and `async` invokes it,

providing a callback `cb`. Code within our function `k` then will invoke `cb` when, and only when, we have a result of the asynchronous computation.[22]

It's possible to use `IO.async` to specify a completely synchronous computation by immediately computing the result and passing it to the callback:

```
def synchronousSum(l: Int, r: Int): IO[Int] =
  IO.async { cb =>
    cb(Right(l + r)) ①
  }
```

① We immediately provide a result to `cb`; we're not starting any asynchronous processing.

But that's a rather artificial example—let's instead create an actual asynchronous effect.

6.1.1. Tracing an asynchronous execution

To demonstrate `IO.async` let's create a new asynchronous `IO` value that uses some callback-based API—in this case, `Future`. We'll reproduce what `IO.fromFuture` does to adapt to the `Future` type using `IO.async`:

Example 26. Using `async` *to adapt a* `Future` *to* `IO`*.*

```
trait API {
  def compute: Future[Int] = ??? ①
}

def doSomething[A](api: API)(implicit ec: ExecutionContext): IO[Int] = {
  IO.async[Int] { cb => ②
    api.compute.onComplete {
      case Failure(t) => cb(Left(t))  ③
      case Success(a) => cb(Right(a)) ③
    }
  }.guarantee(IO.shift) ④
}
```

① Our API returns a `Future`. For this example it's not important how the `Future` is created or what it does, so we mark it unimplemented. However, it *is* important to note that the returned `Future` has been scheduled at this point, in contrast with lazily-executed `IO` values.

② `IO.async` provides a callback `cb` so the API can signal the result of the computation.

③ When the API computes the result it provides it to the callback.

④ The callback may be executing on some non-`IO` thread, so we guarantee that the *next* effect to execute will be on our own threads by `shift`-ing onto our context.

Let's walk through what happens when we execute an effect built with `IO.async`:

```
val api = new API { ... }
val ds = doSomething(api)

ds.unsafeRunSync()
```

In the description below we refer to the block given to `IO.async` as `k`. If it were explicitly written out it would look like:

```
val k: (Either[Throwable,Int] => Unit) => Unit =
  cb => api.compute.onComplete {
    case Failure(t) => cb(Left(t))
    case Success(a) => cb(Right(a))
  }
```

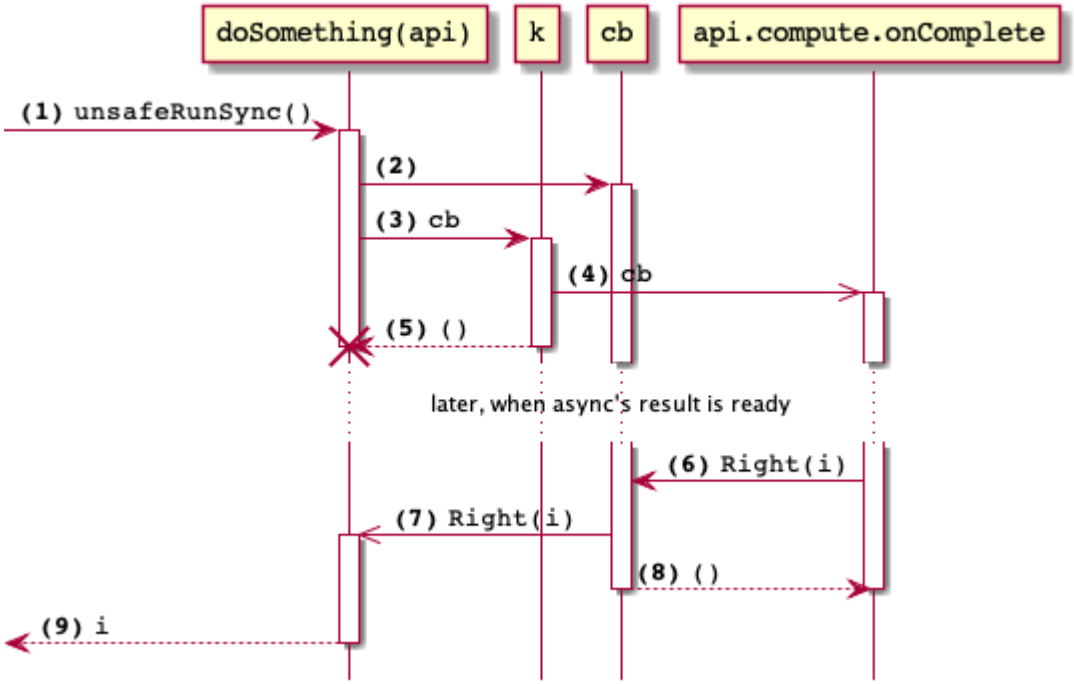

Figure 10. Execution sequence of asynchronous callbacks.

We can then see the order in which the various pieces are invoked:

1. We invoke `unsafeRunSync`, asking the `IO[Int]` created by `doSomething(api)` to compute a value of type `Int`.
2. The `IO` value creates a new callback `cb` that will be invoked by some asynchronous code. `cb` has type `Either[Throwable, Int] ⇒ Unit`.
3. We invoke `k`, passing it `cb`.
4. `k` invokes the asynchronous API, ensuring `cb` will be invoked when the `Future` returned by `api.compute` completes.
5. We've started the asynchronous computation, so `k` completes, returning a `Unit`. The current thread is now *blocked* until `cb` is invoked by the asynchronous computation.
6. The asynchronous computation succeeds in computing an `Int` and invokes `cb(Right(i))`. If the computation failed, it would invoke `cb` with a `Left` containing an exception.
7. `cb` notifies the `unsafeRunSync` thread to *unblock*, passing it the `Right(i)` value (or `Left(ex)` if it had failed).
8. *On the asynchronous API's thread*, `cb` completes, returning a `Unit`.
9. The successful `Int` value is returned, completing the `unsafeRunSync` call.

A bit complex, but not eye-melting! The main idea is to pass around the `cb` callback which is eventually given the result, which then, in this case, can unblock the waiting thread.

Exercise 4: `java.util.concurrent.CompletableFuture`

Let's use `IO.async` to adapt a `java.util.concurrent.CompletableFuture` into an `IO` value.

Code available at `asynchrony/AsyncCompletable.scala`.

```
package com.innerproduct.ee.asynchrony

import cats.effect._
import com.innerproduct.ee.debug._
import java.util.concurrent.CompletableFuture
import scala.jdk.FunctionConverters._

object AsyncCompletable extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    effect.debug.as(ExitCode.Success)

  val effect: IO[String] =
    fromCF(IO(cf()))

  def fromCF[A](cfa: IO[CompletableFuture[A]]): IO[A] =
    cfa.flatMap { fa =>
      IO.async { cb =>
        val handler: (A, Throwable) => Unit = ??? ①

        fa.handle(handler.asJavaBiFunction) ②

        ()
      }
    }

  def cf(): CompletableFuture[String] =
    CompletableFuture.supplyAsync(() => "woo!") ③
}
```

① Write a handler to invoke the callback `cb` when the `CompletableFuture` finishes.

② `handle` executes the handler in the *current* thread, which will be the one executing the function argument to `IO.async`.[23]

③ This executes the given function in the `ForkJoinPool.commonPool` thread pool.

Solution to Exercise

Exercise 5: Never!

`IO.async` provides a callback to report the computation of a successful value or an error. Can we implement an effect that *never* completes?

Code available at `asynchrony/Never.scala`.

```
package com.innerproduct.ee.asynchrony

import cats.effect._
import com.innerproduct.ee.debug._

object Never extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    never
      .guarantee(IO("i guess never is now").debug.void)
      .as(ExitCode.Success)

  val never: IO[Nothing] =
    IO.async(???) ①
}
```

① Implement an effect with `IO.async` that never completes (with a value or a failure).

Solution to Exercise

6.2. Integrating with `Future`

`scala.concurrent.Future` is the most common legacy data type for asynchronous computation in Scala, and as we've seen we can use `IO.async` to implement an `IO` value in terms of an asynchronously executing `Future`. Since it's so common, Cats Effect provides a built-in method: `IO.fromFuture`:

```
def asFuture(): Future[String] = ①
  Future.successful("woo!")

val asIO: IO[String] =
  IO.fromFuture(IO(asFuture)) ②
```

① Let's integrate a method that returns a `Future`. In real code this method would live in some API defined elsewhere.

② `IO.fromFuture` translates a `Future[A]`, itself wrapped in an `IO`, into an `IO[A]`.

Exercise 6: Why does `IO.fromFuture` require a `Future` inside an `IO`?

`IO.fromFuture` has the type signature:

```
def fromFuture[A](iof: IO[Future[A]])(implicit cs: ContextShift[IO]): IO[A]
```

Why does the argument `iof` have type `IO[Future[A]]`, instead of simply having type `Future[A]`?

Hint: we discussed a related idea in Section 1.3.2, "Example: Is `Future` an effect?".

Solution to Exercise

6.3. Summary

1. `IO.async` allows us to build effects that (1) can start asynchronous processes; (2) can emit one result on completion or can end in error.
2. Asynchronous effects fundamentally rely upon continuation passing, where the actual asynchronous computation is given code to run when the computation completes.
3. `scala.concurrent.Future` is a common source of asychronous computation. `IO.fromFuture` transforms a `Future` into a referentially-transparent effect.

[22] Why is the argument to `async` named `k`? It stands for *continuation*, which is traditionally abbreviated with a `k`. (A nod to some German roots in logic?) In any case, the argument to `async` can be viewed what to do next, when the result is available: the computation "continues" once the result is supplied.

[23] If registered with `handleAsync`, then `handler` would execute in the `CompletableFuture`'s thread pool, and subsequent `IO` effects would also executed there, which is probably not the expected behavior. If you use `handleAsync`, then you need to subsequently `IO.shift` after the `IO.async` call to ensure that the next effects occur in the `IO` context.

Chapter 7. Managing resources

It's often the case that state needs to be *managed*. For example, a thread pool needs to be allocated and configured before it can be used, and once we're done with it the threads need to be shut down. Many other kinds of state have the same need for lifecycle management, like:

- *A network connection* maintains a connection to a remote system over some socket networking abstraction. Allocating sockets may be expensive, in addition to the time needed to actually establish a (remote) connection. And those sockets need to be reclaimed when they aren't needed anymore.
- *A database connection*, like a network connection, also needs to talk to a remote system, and will have similar costs like the previous example. It may also manage its own, additional resources, such as threads, involved in the connection protocol.

In Cats Effect, the `Resource` data type represents this *acquire-use-release* pattern to manage state. We'll explore how to create our own `Resource` values, how to compose them, and then learn how to use them in our applications for dependency management.

7.1. Creating a `Resource` to manage state

Let's create a `Resource` to better understand how to manage the lifecycle of some trivial state. We'll use `Resource.make`. It takes two effectful arguments: one to produce (*acquire*) the state, and another to *release* it:

```
def make[A](acquire: IO[A])(release: A => IO[Unit]): Resource[IO, A]
```

We'll then `use` the `Resource` in a program:

Example 27. Making and using a basic `Resource`*. Code available at* `resources/BasicResource.scala`*.*

```
package com.innerproduct.ee.resources

import cats.effect._
import com.innerproduct.ee.debug._

object BasicResource extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    stringResource
      .use { s => ②
        IO(s"$s is so cool!").debug
      }
      .as(ExitCode.Success)

  val stringResource: Resource[IO, String] = ①
    Resource.make(
      IO("> acquiring stringResource").debug *> IO("String")
    )(_ => IO("< releasing stringResource").debug.void)
}
```

① We build a `Resource[IO, String]` with `Resource.make`, passing it two effectful functions. In this example, we add additional `debug` logging to see when these "lifecycle" effects execute.

② The `use` method supplies the created `String` to the given function which must return an `IO` value. Again we `debug` log to see what `String` value was passed.

Running `BasicResource` produces the output:

```
[ioapp-compute-0] > acquiring stringResource
[ioapp-compute-0] String is so cool!
[ioapp-compute-0] < releasing stringResource
```

As you can see, we first acquire the value, then it is used, and then it is released.

It's important to note that `Resource` doesn't perform any effects itself. When we `use` it, it produces a new effect. You can think of `Resource` as a tiny DSL (domain-specific language) to *describe* resource management in terms of individual `IO` effects and the `use` method then "compiles" those instructions into a single `IO` value.

Additionally the `release` effect will not only be executed if the `use` effect completes successfully, but also if it raises an error:

Example 28. A resource is properly released even when the `use` *effect fails. Code available at* `resources/BasicResourceFailure.scala`.

```
package com.innerproduct.ee.resources

import cats.effect._
import com.innerproduct.ee.debug._

object BasicResourceFailure extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    stringResource
      .use(_ => IO.raiseError(new RuntimeException("oh noes!"))) ①
      .attempt
      .debug
      .as(ExitCode.Success)

  val stringResource: Resource[IO, String] =
    Resource.make(
      IO("> acquiring stringResource").debug *> IO("String")
    )(_ => IO("< releasing stringResource").debug.void)
}
```

① Inside the `use` effect we raise an error.

Running `BasicResourceFailure` produces the following output, which shows that no matter what happens during the `use` effect, the `Resource` release effect always executes:

```
[ioapp-compute-0] > acquiring stringResource
[ioapp-compute-0] < releasing stringResource
[ioapp-compute-0] Left(java.lang.RuntimeException: oh noes!)
```

7.1.1. Example: Ensuring a file handle is closed

Here's a more practical example of defining a `Resource` to read from an open file, with the usual post-condition of ensuring the file is closed:[24]

Example 29. Using a `Resource` *to hide and manage the state of a file to be read. Code available at* `resources/FileBufferReader.scala`.

```
package com.innerproduct.ee.resources

import cats.effect._
import java.io.RandomAccessFile

class FileBufferReader private (in: RandomAccessFile) { ①
  def readBuffer(offset: Long): IO[(Array[Byte], Int)] = ②
    IO {
      in.seek(offset)

      val buf = new Array[Byte](FileBufferReader.bufferSize)
      val len = in.read(buf)

      (buf, len)
    }

  private def close: IO[Unit] = IO(in.close()) ③
}

object FileBufferReader {
  val bufferSize = 4096

  def makeResource(fileName: String): Resource[IO, FileBufferReader] = ④
    Resource.make {
      IO(new FileBufferReader(new RandomAccessFile(fileName, "r")))
    } { res =>
      res.close
    }
}
```

① Our `Resource` will manage a hidden `java.io.RandomAccessFile` wrapped in the `FileBufferReader` data type.

② `FileBufferReader` exposes only one method: an effect to read a buffer from an offset.

③ Since we want the `Resource` to manage the hidden state, we make the `close` method inaccessible from outside callers.

④ We make our `Resource` by creating the `FileBufferReader` in an `IO` effect, ensuring that we `close` the state when the `Resource` is released.

7.1.2. Example: Canceling a background task

A perhaps less obvious use of a `Resource` is to manage the lifecycle of a background

task. For example, we may want to fork some (often non-terminating) effect, and later cancel it when it isn't required to run anymore. This suggests we can combine use of a `Fiber` with a `Resource`, where the `Resource` effects are defined as:

acquire `start` the task, producing a `Fiber`

release `cancel` the `Fiber`

Then the lifetime of the background task would directly correspond to the execution of the `use` effect of the `Resource`:

Example 30. Scoping the lifetime of a background task with `Resource`. *Source at* `resources/ResourceBackgroundTask.scala`.

```
package com.innerproduct.ee.resources

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._
import scala.concurrent.duration._

object ResourceBackgroundTask extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    for {
      _ <- backgroundTask.use { _ =>
        IO("other work while background task is running").debug *>
        IO.sleep(200.millis) *>
        IO("other work done").debug ①
      }
      _ <- IO("all done").debug
    } yield ExitCode.Success

  val backgroundTask: Resource[IO, Unit] = {
    val loop =
      (IO("looping...").debug *> IO.sleep(100.millis))
        .foreverM ②

    Resource
      .make(IO("> forking backgroundTask").debug *> loop.start)( ③
        IO("< canceling backgroundTask").debug.void *> _.cancel ④
      )
      .void ⑤
  }
}
```

① The background task will only be running during our `use` effect. We sleep a little bit, to ensure the background task does some work.

② The background task itself is a loop that prints and sleeps. We use the `foreverM` combinator, which is equivalent to

```
- val loop: IO[Nothing] = step.flatMap(_ => loop)
+ val loop: IO[Nothing] = step.foreverM
```

③ The acquire effect forks a `Fiber` and…

④ … the release effect cancels it.

⑤ In this example we don't give the user of the `Resource` access to the `Fiber`,

although you could imagine where this may be useful.

`ResourceBackgroundTask` outputs:

```
[ioapp-compute-0] > forking backgroundTask ①
[ioapp-compute-1] looping...
[ioapp-compute-0] other work while background task is running
[ioapp-compute-2] looping...
[ioapp-compute-3] looping...
[ioapp-compute-4] other work done ②
[ioapp-compute-4] < canceling backgroundTask ③
[ioapp-compute-4] all done
```

① Our effect is forked as a `Fiber`.

② Once the `use` effect finishes...

③ ... the `Fiber` is canceled.

Since this "background task" pattern is common, Cats Effect defines the `background` method on an `IO`:

```
def background: Resource[IO, IO[A]] ①
```

① The resource "value" is an `IO[A]`, which is an effect which lets you join the effect running in the background; it's literally the `join` method of the `Fiber` that the `Resource` manages.

Our code from the example—with `debug` effects removed—could be rewritten as:

```
- Resource.make(loop.start)(_.cancel)
+ loop.background
```

The `background` method corrects a latent problem with manually managing `Fibers`—they may "leak" if they aren't properly canceled. For example:

```
def leaky[A, B](ia: IO[A], ib: IO[B]): IO[(A, B)] =
  for {
    fiberA <- ia.start
    fiberB <- ib.start
    a <- fiberA.join ①
    b <- fiberB.join
  } yield (a, b)
```

① If `ia` raises an error, `fiberA.join` will fail and `fiberB` will still be allocated and running.

7.2. Composing managed state

We can define individual `Resource` values, but building a `Resource` from another `Resource`? No problem, we can compose in multiple ways.

`Resource` is a *functor*; we can `map` over it:

```
val resA: Resource[IO, A] = ???
val resB: Resource[IO, B] = resA.map(makeB)

def makeB(a: A): B = ???
```

`Resource` is an *applicative*; we can `mapN` over two or more values:

```
val resD: Resource[IO, D] =
  (resB, resC).mapN(makeD)

def makeD(b: B, c: C): D = ???
```

`Resource` is a *monad*; we can `flatMap` over it, or more conveniently, we can use a `for`-comprehension:

```
val resC: Resource[IO, C] =
  for {
    a <- resA
    c <- makeC(a)
  } yield c ①

def makeC(a: A): Resource[IO, C] = ???
```

① Alternatively we could write `resA.flatMap(makeC)`.

If we add another `Resource` to the `BasicResource` example, we can compose them together to see the order in which the lifecycle effects are executed:

Example 31. Composing multiple resources. Code available at `resources/BasicResourceComposed.scala`.

```
package com.innerproduct.ee.resources

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._

object BasicResourceComposed extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    (stringResource, intResource).tupled ②
      .use {
        case (s, i) => ②
          IO(s"$s is so cool!").debug *>
          IO(s"$i is also cool!").debug
      }
      .as(ExitCode.Success)

  val stringResource: Resource[IO, String] =
    Resource.make(
      IO("> acquiring stringResource").debug *> IO("String")
    )(_ => IO("< releasing stringResource").debug.void)

  val intResource: Resource[IO, Int] = ①
    Resource.make(
      IO("> acquiring intResource").debug *> IO(99)
    )(_ => IO("< releasing intResource").debug.void)
}
```

① We create another `Resource`, this time a managed `Int` value.

② We compose `stringResource` with `intResource` via `tupled`, which is equivalent to `mapN((s, i) ⇒ (s, i))`, to produce a `Resource[IO, (String, Int)]`. We then can deconstruct the tuple in the function passed to the `use` method.

Running `resources/BasicResourceComposed.scala`, we see:

```
[ioapp-compute-0] > acquiring stringResource
[ioapp-compute-0] > acquiring intResource
[ioapp-compute-0] String is so cool!
[ioapp-compute-0] 99 is also cool!
[ioapp-compute-0] < releasing intResource    ①
[ioapp-compute-0] < releasing stringResource ①
```

① Note that the resources are released in the *opposite* order in which they are acquired!

7.2.1. Parallel resource composition

`Resource` is a monad, so we can compose it with various combinators. But it also has a `Parallel` typeclass instance, so resource management can occur in parallel rather than sequentially!

All we need to do is use the `par`-prefixed combinators. For example, in the `BasicResourceComposed` application above, we replace `tupled` with `parTupled`:

```
- (stringResource, intResource).tupled
+ (stringResource, intResource).parTupled
```

and the resources are initialized and cleaned-up on their own threads:

```
[ioapp-compute-2] > acquiring intResource    ①
[ioapp-compute-1] > acquiring stringResource ①
[ioapp-compute-1] String is so cool!
[ioapp-compute-1] 99 is also cool!
[ioapp-compute-3] < releasing stringResource ②
[ioapp-compute-4] < releasing intResource    ②
```

① Resource initialization in parallel.

② Resource clean-up in parallel.

Exercise 7: Early-release of Resources

Here we have a more typical use of resources for an application: we load an external configuration that contains the connection string to a database. We then connect to the database to issue some queries.

We load the configuration data using a managed `scala.io.Source`. At the same time, if we have a loaded `Config` value, we can then open a managed `DbConnection` value to then use it for queries.

The current program shows a problem:

```
[ioapp-compute-0] > opening Source to config
[ioapp-compute-0] read Config(exampleConnectURL) ①
[ioapp-compute-0] > opening Connection to exampleConnectURL
[ioapp-compute-0] (results for SQL "SELECT * FROM users WHERE id = 12")
[ioapp-compute-0] < closing Connection to exampleConnectURL
[ioapp-compute-0] < closing Source to config ②
```

① Once we read the configuration from the `Source`, we can close the `Source`.

② But due to the nesting properties of `Resource` the `Source` is only closed after the `DbConnection` is closed. The `Source` is kept open too long!

Can you modify the `configResource` so it closes the `sourceResource` once the `Config` value is available?

We'd like the output of the program to look like this instead:

```
[ioapp-compute-0] > opening Source to config
[ioapp-compute-0] read Config(exampleConnectURL)
[ioapp-compute-0] < closing Source to config ①
[ioapp-compute-0] > opening Connection to exampleConnectURL
[ioapp-compute-0] (results for SQL "SELECT * FROM users WHERE id = 12")
[ioapp-compute-0] < closing Connection to exampleConnectURL
```

① We want to close the `Source` as soon as we can.

Code available at `resources/EarlyRelease.scala`.

```
package com.innerproduct.ee.resources

import cats.effect._
import com.innerproduct.ee.debug._
import scala.io.Source

object EarlyRelease extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
```

```
    dbConnectionResource
      .use { conn =>
        conn.query("SELECT * FROM users WHERE id = 12").debug
      }
      .as(ExitCode.Success)

  val dbConnectionResource: Resource[IO, DbConnection] =
    for {
      config <- configResource
      conn <- DbConnection.make(config.connectURL)
    } yield conn

  lazy val configResource: Resource[IO, Config] = ①
    for {
      source <- sourceResource
      config <- Resource.liftF(Config.fromSource(source)) ②
    } yield config

  lazy val sourceResource: Resource[IO, Source] =
    Resource.make(
      IO(s"> opening Source to config")
        .debug *> IO(Source.fromString(config))
    )(source => IO(s"< closing Source to config").debug *> IO(source.close))

  val config = "exampleConnectURL"
}

case class Config(connectURL: String)

object Config {
  def fromSource(source: Source): IO[Config] =
    for {
      config <- IO(Config(source.getLines().next()))
      _ <- IO(s"read $config").debug
    } yield config
}

trait DbConnection {
  def query(sql: String): IO[String] // Why not!?
}

object DbConnection {
  def make(connectURL: String): Resource[IO, DbConnection] =
    Resource.make(
      IO(s"> opening Connection to $connectURL").debug *> IO(
        new DbConnection {
          def query(sql: String): IO[String] =
            IO(s"""(results for SQL "$sql")""")
        }
      )
    )(_ => IO(s"< closing Connection to $connectURL").debug.void)
}
```

① Can you modify the `configResource` so it closes the `sourceResource` once the `Config` value is available?

② `Resource.liftF` transforms an `F[A]` into a `Resource[F, A]`, where there isn't any allocation or release effect, only a value of type `A` produced by the `F[A]` and subsequently used by the `use` method of the `Resource`.

Solution to Exercise

7.3. Resources for dependency management

Most of the code we've seen so far have been short examples that demonstrate focused topics like parallelism, concurrency, and so on. While these examples have been written using `IOApp`, we haven't discussed how to build larger applications with `IOApp`, or what issues may arise as we try to compose sets of effects into a larger program.

Since a `Resource` perfectly encapsulates the effectful allocation and clean-up of a value, we can use it to manage our application's dependencies for us. Our `IOApp`-based application will then be structured into three distinct concerns:

1. Dependency lifecycles managed by a single, possibly composed, `Resource`.
2. Application logic that uses the dependencies—without any concern over their lifecycle.
3. The top-level of the application initiates the allocation of the dependencies, their use by the logic, and then cleans them up.

We've already been using this structure in the previous examples, but let's explicitly call out the concerns in another example `IOApp`-based application:

Example 32. Structuring an `IOApp` *into dependencies and logic. Code available at* `resources/ResourceApp.scala`.

```
package com.innerproduct.ee.resources

import cats.effect._
import cats.implicits._

object ResourceApp extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    resources ①
      .use { ③
        case (a, b, c) =>
          applicationLogic(a, b, c) ②
      }
      .as(ExitCode.Success)

  val resources: Resource[IO, (DependencyA, DependencyB, DependencyC)] = ①
    (resourceA, resourceB, resourceC).tupled

  val resourceA: Resource[IO, DependencyA] = ???
  val resourceB: Resource[IO, DependencyB] = ???
  val resourceC: Resource[IO, DependencyC] = ???

  def applicationLogic( ②
      a: DependencyA,
      b: DependencyB,
      c: DependencyC
  ): IO[ExitCode] =
    ???
  }

trait DependencyA
trait DependencyB
trait DependencyC
```

① We compose a set of managed dependencies into a *single* `Resource` value.

② The application logic uses the dependencies—but does not manage them.

③ At the beginning of our application we `use` our managed dependencies, providing them to the application logic. The dependencies only exist within the scope of the `use` block.

7.4. Summary

1. The `Resource` data type captures the pattern where the code for state acquisition and release is separated from code that *uses* the state. A `Resource`

can be composed into other `Resource` values, both serially and in parallel.

2. We can use `Resource` to represent the lifecycle of our application dependencies. We then `use` them in our `IOApp` to acquire them during the execution of the dependent code, and to ensure they are released.

[24] Code stolen (and subsequently adapted and simplified) from Fabio Labella in the Cats Effect Gitter chat room at https://gitter.im/typelevel/cats-effect?at=5ed132c1ff7a920a721d1402.

Chapter 8. Testing effects

Testing is a tremendously complicated and nuanced subject. And since we've been discussing effects like `IO`, which can encapsulate side effects that *by definition* can't be observed, testing anything involving `IO` is a very open-ended proposition.

Instead we're going to focus on two areas, one fairly simple and the other rather complicated: testing the *values* produced by an `IO` effect, and controlling how `IO` effects interact with their runtime dependencies like `ExecutionContext` so we can make assertions about "when" their execution occurs.

8.1. Assertions on effectful values

Since the very idea of `IO` is that we delay its execution, to test an `IO` value means we need to run it so we can then make assertions about the value it produces. That means we need to invoke one of the `unsafe` methods of `IO`, like `unsafeRunSync`:

```
def assertGreaterThanZero(i: IO[Int]) =
  assert(i.unsafeRunSync() > 0) ①
```

① We're deliberately avoiding the choice of assertion mechanisms by using the built-in `assert` method of Scala. You can use your favorite testing or "matchers" library.

Alternatively you could make the assertion "inside" the `IO` and ensure the composed effect is executed:

```
def assertGreaterThanZero(i: IO[Int]) =
- assert(i.unsafeRunSync() > 0)
+ i.map(j => assert(j > 0)).unsafeRunSync()
```

Remember, `unsafeRunSync` will throw an exception if the effect fails or is cancelled. If your testing framework doesn't treat thrown exceptions as failures, or you want to assert that a failure or cancellation *has* happened, you can use `attempt` to lift the success value or failure/cancelation exception into a successful `Either` value:

```
def assertUnsuccessful[A](ia: IO[A]) =
  assert(ia.attempt.unsafeRunSync().isLeft)
```

8.1.1. Faking effects with interfaces

By executing the `IO` you cause it to perform its effects, but what if during testing you don't want the actual effect to happen? For example, an effect could be

sending an email or updating a database. This is a very common concern for any kind of testing, and the most common solution is to ensure that the effects we want to avoid are isolated behind an interface that we can provide an alternate implementation for. Instead of a concrete method that returns an effect—a method we can't override in order to "control" it—we abstract over the effectful operation so we can.

For example, instead of directly creating an effect to send an email, we'll invoke a method on an interface to send it:

```
- def send(to: EmailAddress, email: Email): IO[Unit] = ???
+ trait EmailDelivery {
+   def send(to: EmailAddress, email: Email): IO[Unit]
+ }
```

Along with a "real" implementation, we could then create "fake" ones for testing where we control the behavior. We could create an instance of the `EmailDelivery` trait that always fails:

```
class FailingEmailDelivery extends EmailDelivery {
  def send(to: EmailAddress, email: Email): IO[Unit] =
    IO.raiseError(new RuntimeException(s"couldn't send email to $to"))
}
```

We then use our faked implementation during the testing of the code that uses it. For example we may be testing the behavior of a user registration service which uses our `EmailDelivery` for additional effects:

```
class UserRegistration(emailDelivery: EmailDelivery) { ①
  def register(email: EmailAddress): IO[Unit] =
    for {
      _ <- save(email)
      _ <- emailDelivery.send(to, new Email(???))
    } yield ()

  private def save(email: EmailAddress): IO[Unit] = ???
}
```

① We pass the interface as a dependency so we can choose a real or fake implementation.

A very basic test might assert that “registration should fail if the registration email could not be sent”:

```
def registrationFailsIfEmailDeliveryFails(email: EmailAddress) =
  new UserRegistration(new FailingEmailDelivery)
    .send(email)
    .attempt
    .map(result => assert(result.isLeft, s"expecting failure, but was $result"))
    .unsafeRunSync
```

8.2. Testing effect scheduling by controlling its dependencies

Faking effects with interfaces works well to increase the modularity and testability of our own code, but what about testing aspects of Cats Effect itself? Cats Effect itself uses the same technique: the `TestContext`[25] helper class lets us use faked `ExecutionContext` and `Timer` instances in the effectful code we want to test, and then explicitly control the effect scheduling in our tests. We can then make assertions about the relative execution order of effects.

Here we instantiate a `TestContext` and bring its members into scope so our effects can reference them:

```
import cats.effect.laws.util.TestContext

val ctx = TestContext() ①

implicit val cs: ContextShift[IO] = ctx.ioContextShift ②
implicit val timer: Timer[IO] = ctx.timer ②
```

① Instantiating a `TestContext`.

② Bringing the context’s `ContextShift` and `Timer` into scope.

In our tests we then can advance the effect scheduling clock manually:

```
val timeoutError = new TimeoutException
val timeout = IO.sleep(10.seconds) *> IO.raiseError[Int](timeoutError) ①
val f = timeout.unsafeToFuture() ②

// Not yet
ctx.tick(5.seconds)          ③
assertEquals(f.value, None) ③

// Good to go:
ctx.tick(5.seconds)                                  ④
assertEquals(f.value, Some(Failure(timeoutError))) ④
```

① The `timeout` effect is what we want to test: we should see an error only after ten seconds has elapsed.

② In order to execute the effect, we convert it to a `Future`, which schedules it to be run with our controllable `ExecutionContext`. But time doesn't move forward until we say it does!

③ Here we advance the clock five seconds, and assert that our effect hasn't produced a value yet.

④ After another five seconds, our ten second sleep should have expired and caused the error to be raised. We then assert the effect produced an error.

8.3. Summary

1. Asserting conditions on effects requires them to be executed.
2. To test computations that use effects, we can "fake" those effects by abstracting an interface over their creation.
3. To make assertions about effect execution order we can use `TestContext` from `cats.effect.laws`. We then schedule the effects for execution—by transforming them to `Future`--and subsequently advance our "clock" to assert "when" effects happen.

[25] `TestContext` resides in the `cats.effect.laws` package, which is *not* part of the core Cats Effect dependency. The sbt dependency for the "laws" module would be `"org.typelevel" %% "cats-effect-laws" % "2.1.3"`.

Chapter 9. Concurrent coordination

So far we've composed multiple effects so they run concurrently and in parallel. We noted that, at first, effects only allow us to talk about the values they produce, and we can create new effects by composing the outputs of other effects. We then added the notion of a `Fiber` to represent an already-executing effect, and with it we can start to control concurrent effects by joining or canceling them. But we haven't yet discussed *coordination* between concurrent effects.

By coordination, we mean the behavior of one effect should depend on another. For example, how can we share state between effects when that state might be concurrently updated? Or how can we ensure one effect only proceeds once work is complete in another?

We'll discuss the first issue by using the `Ref` data type for sharing mutable state. We'll then show how the `Deferred` data type can provide concurrent effect serialization without blocking any actual threads. Finally we'll model even more complex behavior by composing these two concurrency primitives together to form a concurrent state machine.

9.1. Atomic updates with `Ref`

The first issue we are going to address is *how do we safely share "mutable" state?* To demonstrate this, we'll share a counter between concurrently running effects. In this case, one effect will increment the counter whenever the effect runs, while the other will periodically print the value of the counter.

Let's build our effects and use a `var` to share the counter state between them:

Example 33. Concurrent effects sharing state via a `var`. *Code available at* `coordination/ConcurrentStateVar.scala`.

```
package com.innerproduct.ee.coordination

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._
import scala.concurrent.duration._

object ConcurrentStateVar extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    for {
      _ <- (tickingClock, printTicks).parTupled ①
    } yield ExitCode.Success

  var ticks: Long = 0L ②

  val tickingClock: IO[Unit] =
    for {
      _ <- IO.sleep(1.second)
      _ <- IO(System.currentTimeMillis).debug
      _ = (ticks = ticks + 1) ③
      _ <- tickingClock
    } yield ()

  val printTicks: IO[Unit] =
    for {
      _ <- IO.sleep(5.seconds)
      _ <- IO(s"TICKS: $ticks").debug.void ④
      _ <- printTicks
    } yield ()
}
```

① We start the two effects in parallel.

② We use a `var` to hold our state.

③ `tickingClock` prints the current time every second, and here updates the `ticks` counter.

④ Every five seconds `printTicks` awakens and prints the current value of the `ticks` counter.

Running this outputs:

```
[ioapp-compute-0] 1600818291481
[ioapp-compute-1] 1600818292492
[ioapp-compute-2] 1600818293497
[ioapp-compute-3] 1600818294503
[ioapp-compute-4] TICKS: 4
[ioapp-compute-5] 1600818295508
[ioapp-compute-6] 1600818296514
[ioapp-compute-7] 1600818297515
[ioapp-compute-0] 1600818298516
[ioapp-compute-1] 1600818299521
[ioapp-compute-2] TICKS: 9
[ioapp-compute-3] 1600818300522
```

...and so on. We see the clock advance every one thousand milliseconds or so, and we see `TICKS` printed every five seconds.

You may have been taught that sharing mutable state is "bad". What issues are there? In the previous example, there was only one effect that could update the state. But what if there were multiple `tickingClock` effects running, each updating the same `var`? Would we always be correctly incrementing the state? One can imagine that *two* concurrent `tickingClock` effects could execute like this:

```
// tickingClock #1
ticks =
  ticks ①
                    // tickingClock #2
                    ticks = ticks + 1 ②
    + 1 ③
```

① Clock #1 reads the current value of `ticks`, which is `0`.

② Meanwhile clock #2 increments `ticks`, which is now 1.

③ Clock #1 now increments the previously read value, which was `0`, and updates `ticks` to 1. We didn't record the increment of clock #2!

With multiple writers, we can have what is called the *lost update problem*.[26] That is to say, is updating the state *atomic*? For a `var`, the answer is "no".

Luckily this is a solved problem. Cats Effect has `Ref`, which wraps the Java data type `AtomicReference`. With `Ref`, updates to state are *atomic*.

To create a `Ref`, we use the `Ref[IO].of` factory method, passing it the initial value to store. For example:

```
val ticks: IO[Ref[IO, Long]] =
  Ref[IO].of(0L) ①
```

① Initialize the `Ref` with the value zero.

The factory method returns the `Ref` as an effect—this is an important detail. Consider this code:

```
val doSomething =
  for {
    ref <- ticks ①
    // do something with `ref`
  } yield ()

val twice = (doSomething, doSomething).tupled
```

① The created `Ref` is scoped within the `doSomething` method.

If you were expecting `twice`, which invokes `doSomething` twice, to (somehow) act on the same `Ref`, that's not what would happen. In this case, `doSomething` *creates* a `Ref` and uses it, but that `Ref` only exists within the `doSomething` scope. Two distinct `Ref` values would be created during the two `doSomething` effects. So whenever we want to share state like a `Ref`, we need to share the actual `Ref`, not the effect that "creates" it:

```
def doSomething(ref: Ref[IO, Long]) = ①
  for {
    // do something with `ref`
  } yield ()

val twice =
  for {
    ref <- ticks ②
    _ <- (doSomething(ref), doSomething(ref)).tupled
  } yield ()
```

① Shared state is provided to, and not created within, the current scope.

② The scope of the shared state has expanded to include both invocations of `doSomething`.

Let's update our example by changing the `var` to a `Ref`:

Example 34. Concurrent effects sharing state via a `Ref`. *Code available at* `coordination/ConcurrentStateRef.scala`.

```
package com.innerproduct.ee.coordination

import cats.effect._
import cats.effect.concurrent.Ref ①
import cats.implicits._
import com.innerproduct.ee.debug._
import scala.concurrent.duration._

object ConcurrentStateRef extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    for {
      ticks <- Ref[IO].of(0L) ②
      _ <- (tickingClock(ticks), printTicks(ticks)).parTupled ③
    } yield ExitCode.Success

  def tickingClock(ticks: Ref[IO, Long]): IO[Unit] =
    for {
      _ <- IO.sleep(1.second)
      _ <- IO(System.currentTimeMillis).debug
      _ <- ticks.update(_ + 1) ④
      _ <- tickingClock(ticks)
    } yield ()

  def printTicks(ticks: Ref[IO, Long]): IO[Unit] =
    for {
      _ <- IO.sleep(5.seconds)
      n <- ticks.get ⑤
      _ <- IO(s"TICKS: $n").debug
      _ <- printTicks(ticks)
    } yield ()
}
```

① Note that `Ref` is in the `cats.effect.concurrent` package, separate from the main Cats Effect package.

② We create a new `Ref` initialized to `0`. Creating a `Ref` is also an effect, so we add it into our for-comprehension to be able to refer to the produced value.

③ We share the `Ref` with both effects that will run in parallel.

④ We use the `update` method to *atomically* update the state in `ticks`.

⑤ We access the current state with the `get` method.

It's important to note that we've followed our Effect Pattern again: we've replaced side effects—updating a `var` for example—with safe `IO` effects produced by the `Ref` data type.

9.1.1. Using `Ref`: getting and setting state

Using `Ref` is quite simple. We can `get` the current state:

```
def get(): IO[A]
```

There are a few methods to atomically set the state:

```
def set(value: A): IO[Unit]

def getAndSet(value: A): IO[A] ①
```

① Sets the value to `A` and returns the *previous* value of the `Ref`.

We can also update the current state with a function instead of a value:

```
def update(f: A => A): IO[Unit]

def getAndUpdate(f: A => A): IO[A] ①
def updateAndGet(f: A => A): IO[A] ②
```

① Updates the value with the function `f` and returns the *previous* value.

② Updates the value with the function `f` and returns the *new* value.

You can also update the state but return a value of a different type using `modify`:

```
def modify[B](f: A => (A, B)): IO[B]
```

It's important to note that the `update`-style methods are given a *pure* (side effect-free) function, because it's possible that the function you provide may be run *more than once*. WAT?

(We previously said that updates were atomic, so wouldn't the update function be run only once?)

It *could be* that the implementation of the atomic update only runs the function once: we could *pessimistically* assume there are multiple updates happening, and use some lower-level mutual exclusion mechanism to ensure only one operation is allowed to proceed. Instead, the actual implementation of `Ref` uses an *optimistic* strategy where it assumes only one update is happening concurrently. However if another concurrent update succeeds, our update operation will be retried.

Therefore we *don't* want to perform any side effects with our function.

Let's demonstrate performing a side effect within a `modify` to show it running more than once when multiple writers are racing to "win" the atomic update:

Example 35. Update functions passed to a `Ref` *may get invoked multiple times. Code available at* `coordination/RefUpdateImpure.scala`.

```
package com.innerproduct.ee.coordination

import cats.effect._
import cats.effect.concurrent.Ref
import cats.implicits._

object RefUpdateImpure extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    for {
      ref <- Ref[IO].of(0)
      _ <- List(1, 2, 3).parTraverse(task(_, ref)) ①
    } yield ExitCode.Success

  def task(id: Int, ref: Ref[IO, Int]): IO[Unit] =
    ref
      .modify(previous => id -> println(s"$previous->$id")) ②
      .replicateA(3) ③
      .void
}
```

① We "race" three tasks updating the same `Ref` in parallel via `parTraverse`.

② We perform a side effect (the `println`) as part of the return value of the function passed to `modify`.

③ `replicateA` repeats an effect `n` times.

We expect nine atomic updates (three elements with three `modify` effects each), but see more than nine calls to the function we supply to `modify`!

```
0->1
0->3
1->3
0->2
3->2
2->3
2->2
3->1
3->3
1->1
3->2
1->3
1->2
3->2
2->2
```

We can see this optimistic update strategy in action in a (slightly modified) implementation of `modify`, where the `ar` value below is a Java `AtomicReference` value whose `compareAndSwap` method will update the state only if the first argument is equal to the current state and return that fact as a `Boolean`:

```
def modify(f: A => (A, B)): IO[B] = {
  @tailrec
  def spin: B = {
    val current = ar.get ①
    val (updated, b) = f(current) ②
    if (!ar.compareAndSet(current, updated)) spin ③
    else b ④
  }
  IO.delay(spin)
}
```

① We copy the current value.

② We compute the updated value and result value using `f`.

③ Atomically update `ar`. If the current value is no longer the same as `current` due to a concurrent update, repeat `spin`.

④ Otherwise the `Ref` now has value `updated`, so return value `b`.

To ensure we execute *an effect* only once for a given update, we replace the `println` side effect with an `IO` *value*:

```
ref
- .modify(previous => id -> println(s"$previous->$id"))
+ .modify(previous => id -> IO(s"$previous->$id").debug)
+ .flatten ①
```

① We `flatten` the result of `modify` because we are returning an `IO` from within another `IO`.

We'll then see the expected number (nine) of executed effects:

```
[ioapp-compute-1] 2->1
[ioapp-compute-2] 0->2
[ioapp-compute-3] 1->3
[ioapp-compute-1] 2->1
[ioapp-compute-3] 3->3
[ioapp-compute-1] 1->1
[ioapp-compute-2] 3->2
[ioapp-compute-3] 1->3
[ioapp-compute-2] 3->2
```

9.2. Write-once synchronization with `Deferred`

Here's a different issue: how can we know when our concurrently updated counter is in a particular state? Let's pretend it's *extremely* important to print `"BEEP!"` when the counter from the previous example reaches 13, so we poll the counter every second:

```
def beepWhen13(ticks: Ref[IO, Long]): IO[Unit] = ①
  for {
    t <- ticks.get ②
    _ <- if (t >= 13) IO("BEEP!").debug ③
         else IO.sleep(1.seconds) *> beepWhen13(ticks) ④
  } yield ()
```

① We pass the `Ref` that holds our counter state.

② We get the current value of the counter.

③ If the counter is greater than or equal to 13, we beep and stop.[27]

④ Otherwise we sleep and then recurse to start the process again.

It's a good exercise to think through how long we should wait to check the condition we are interested in—it certainly depends on the true update frequency

of the state. On one hand if we poll too often we're being inefficient, and on the other hand if we poll too infrequently then our reaction to the state change is delayed.

Instead of having to guess a polling interval, we can push this responsibility behind an abstraction that will, from the outside, *block* subsequent execution until the condition is fulfilled. We won't have to do anything. At the same time, we'll push the signaling of the condition closer to the state.

In Cats Effect this abstraction is the `Deferred` data type. Let's replace the previous synchronization that used polling with the blocking `Deferred`:

Example 36. Using `Deferred` *for blocking synchronization. Code available at* `coordination/IsThirteen.scala`.

```
package com.innerproduct.ee.coordination

import cats.effect._
import cats.effect.concurrent._
import cats.implicits._
import com.innerproduct.ee.debug._
import scala.concurrent.duration._

object IsThirteen extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    for {
      ticks <- Ref[IO].of(0L)
      is13 <- Deferred[IO, Unit] ①
      _ <- (beepWhen13(is13), tickingClock(ticks, is13)).parTupled ②
    } yield ExitCode.Success

  def beepWhen13(is13: Deferred[IO, Unit]) =
    for {
      _ <- is13.get ③
      _ <- IO("BEEP!").debug
    } yield ()

  def tickingClock(ticks: Ref[IO, Long], is13: Deferred[IO, Unit]): IO[Unit] =
    for {
      _ <- IO.sleep(1.second)
      _ <- IO(System.currentTimeMillis).debug
      count <- ticks.updateAndGet(_ + 1)
      _ <- if (count >= 13) is13.complete(()) else IO.unit ④
      _ <- tickingClock(ticks, is13)
    } yield ()
}
```

① We create a `Deferred` that will hold a `Unit` value once the condition is met.

② The two effects are only communicating through the shared `is13` value.

③ Calling `get` will block the current effect until `is13` has a value.

④ `tickingClock` is responsible for evaluating the condition we're interested in. If `ticks` has value 13, it calls `complete` to provide a value to the `Deferred`, unblocking any waiting effects.

Such a useful data type with only two methods: `get` and `complete`!

One final issue, however—when execution is blocked from invoking `Deferred.get`, does that mean that the underlying thread is blocked? Fortunately for us, the answer is "no". Instead of blocking execution at the thread level, Cats Effect uses so-called semantic blocking: the effect is suspended at a logical level, but the underlying thread can be reused for executing other concurrent effects.

Synchronization

When we talk about how concurrent effects might coordinate together—by sharing state, and so on—we will need a more precise way of talking about the constraints that coordination requires. In Computer Science the term *synchronization* is used to talk about enforcing a relationship between effects.[28] For example,

serialization: effect `B` should only happen *after* effect `A`; and

mutual-exclusion: effect `A` should *never* happen at the same time as effect `B`

are each a type of synchronization constraint.[29]

How is synchronization expressed with `IO`? Via `flatMap` (or an equivalent `for`-comprehension) we can express the fact that one effect *happens after* another:

```
for {
  a <- effectA
  b <- effectB ①
} yield a + b
```

① `effectB` definitely happens after `effectA`.

In contrast, using `parMapN` expresses no a-priori relationship between effects other than transforming each effect's "output":

```
(effectA, effectB).parMapN((a, b) => a + b)
```

It's *non-deterministic* which effect executes first when using something like `parMapN`, and that's "a feature". No synchronization is required.

What about when we use fibers? What notions of synchronization are present? When we `start` a fiber, we're expressing an independent, concurrent effect. When we `join`, any subsequent effect must happen after the fiber completes.

Let's now consider more complex effects that aren't necessarily easily expressed with `flatMap`, `parMapN`, or by using `Fibers`. What kinds of synchronization do they require?

updating shared state	Updates must be atomic, i.e., mutually-exclusive with respect to other updates, otherwise updates may be “lost”.
reading shared state	Concurrent reads don’t require any synchronization. We read whatever the “current” value is, independent of anything else.
blocking	Subsequent effects must happen after the “blocking” effect “unblocks” (serialization).

9.3. Concurrent state machines

`Ref` and `Deferred` are the building blocks of concurrency. With `Ref` we can ensure atomic updates of shared state, and `Deferred` gives us the ability to serialize the execution of an effect with respect to some newly-produced state. Together we can build larger and more complex concurrent behaviors. One technique to do this is to create a *concurrent state machine*.[30] To build one we:

1. Define an interface whose methods return effects.
2. Implement the interface by building a state machine where:
   a. state (with type `S`) is atomically managed via a `Ref[IO, S]` value;
   b. each interface method is implemented by a state transition function affecting the `Ref`; and
   c. any state-dependent blocking behavior is controlled via `Deferred` values.

As an example, we’ll follow this recipe to build a structure called a *countdown latch*.

9.3.1. Example: countdown latch

The behavior we’d like to model is *to block subsequent effects until a certain number of (possibly concurrent) effects have occurred.*

The metaphor of a *latch* is used because a latch is used to keep a door closed until the latch is opened. The term *countdown* refers to the algorithm for how the latch is opened: a counter is decremented, and when the counter reaches zero, the latch opens.

There are two logical roles that concurrently coordinate through the shared latch:

1. *readers* wait for the latch to open; and
2. *writers* decrement the latch counter.

The latch itself is responsible for "opening" when its counter reaches zero.

Let's fulfill step one of our recipe ("define an interface whose methods return effects") by encapsulating the actions of the two roles as methods on a shared `CountdownLatch` interface:

```
trait CountdownLatch {
  def await: IO[Unit] ①
  def decrement: IO[Unit] ②
}
```

① Readers will `await` the opening of the latch. The caller will be *blocked* and no value will be produced until the latch opens.

② Writers will `decrement` the latch counter, which may open the latch.

A "reader" will be waiting for the latch to open, perhaps denoting a set of prerequisite actions have occurred:

```
  def actionWithPrerequisites(latch: CountdownLatch) =
    for {
      _ <- IO("waiting for prerequisites").debug
      _ <- latch.await ①
      result <- IO("action").debug ②
    } yield result
```

① We block until the `latch` opens.

② Once the `latch` opens, we can run the action.

At the same time, a "writer" is fulfilling one or more of those prerequisites:

```
  def runPrerequisite(latch: CountdownLatch) =
    for {
      result <- IO("prerequisite").debug
      _ <- latch.decrement ①
    } yield result
```

① Once the prerequisite action is completed, we `decrement` the latch.

Other code would run each of these roles concurrently:

```
val prepareAndRun =
  for {
    latch <- CountdownLatch(1)
    _ <- (actionWithPrerequisites(latch), runPrerequisite(latch)).parTupled
  } yield ()
```

It's important to note that the two effects are *only* communicating through the shared `CountdownLatch`. They don't directly know anything about each other.

When we run it we would see output like:

```
[ioapp-compute-1] waiting for prerequisites
[ioapp-compute-2] prerequisite
[ioapp-compute-1] action
```

Let's implement it! A `CountdownLatch` will be in one of two states:

1. *outstanding*: we have `n` outstanding `decrement()` operations to expect; or
2. *done*: we have invoked `decrement()` `n` (or more) times.

We'll encode the state—step 2a of our recipe—as an algebraic data type:

```
sealed trait State
case class Outstanding(n: Long, whenDone: Deferred[IO, Unit]) extends State
case class Done() extends State
```

For each method of the interface, the behavior of the latch will depend on its current state:

When a "reader" calls `await()`

- If our state is `Outstanding(n, whenDone)`, there are `n` outstanding `decrement` calls, so block the caller via `whenDone.get`.
- If our state is `Done()`, do nothing.

When a "writer" calls `decrement()`

- If our state is `Outstanding(n, whenDone)`
  - If `n` is 1, this is the last `decrement()`. Transition to `Done` and unblock any blocked `await()` calls via `whenDone.complete()`.
  - Otherwise decrement `n`.
- If our state is `Done()`, do nothing.

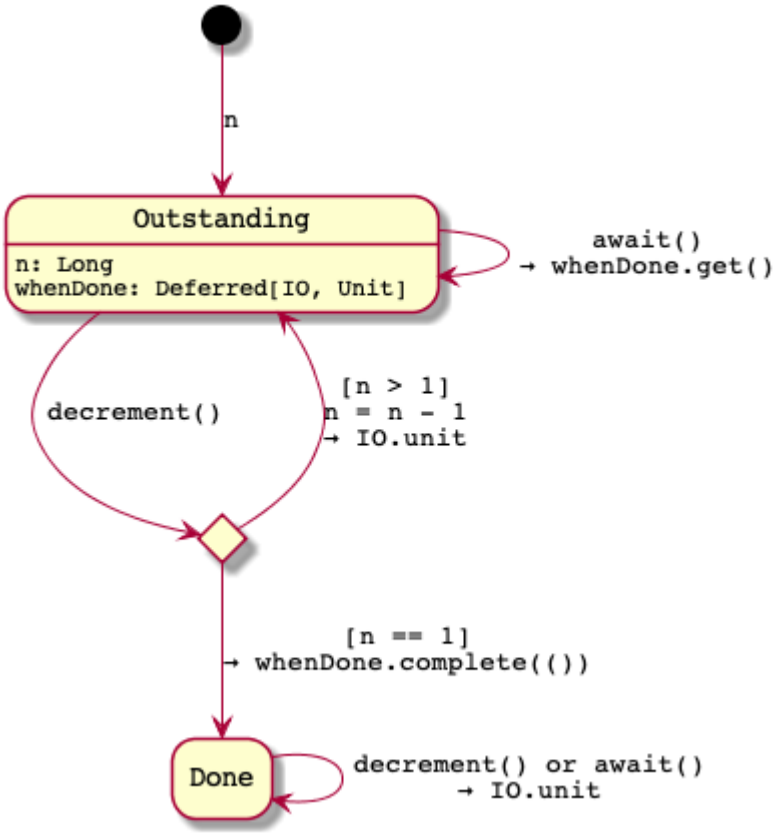

Figure 11. Concurrent state machine for a countdown latch that opens after `n` *events. A* `Ref[IO, State]` *holds the current state.*

When we construct the `CountdownLatch` we'll control concurrent access to the state with a `Ref` and create a `Deferred` to control our blocking behavior. We'll then translate the state transitions into code almost exactly as previously described:

Example 37. `coordination/CountdownLatch.scala`

```
object CountdownLatch {
  def apply(n: Long)(implicit cs: ContextShift[IO]): IO[CountdownLatch] =
    for {
      whenDone <- Deferred[IO, Unit] ①
      state <- Ref[IO].of[State](Outstanding(n, whenDone)) ②
    } yield new CountdownLatch {
      def await: IO[Unit] =
        state.get.flatMap { ③
          case Outstanding(_, whenDone) => whenDone.get ④
          case Done()                   => IO.unit
        }

      def decrement: IO[Unit] =
        state.modify { ⑤
          case Outstanding(1, whenDone) =>
            Done() -> whenDone.complete(()) ⑥
          case Outstanding(n, whenDone) =>
            Outstanding(n - 1, whenDone) -> IO.unit ⑦
          case Done() => Done() -> IO.unit
        }.flatten ⑧
    }
}
```

① We create a `Deferred[IO, Unit]` that we'll use to block and unblock `await()` callers.

② We enforce atomic access to the current state with a `Ref[IO, State]` that we initialize to `Outstanding` with `n` expected decrements.

③ `await()` never changes the state, so we only act on the value from `state.get`.

④ If decrements are outstanding, we return a blocking effect that unblocks when the `Deferred` is completed.

⑤ `decrement()` always changes the state, so we use `Ref.modify`.

⑥ This is the last decrement, so we transition to `Done` and return an effect that completes the `Deferred` to unblock anyone who has invoked `await()`.

⑦ We decrement the counter and return an effect which does nothing.

⑧ Our use of the `state.modify` method returns an `IO[IO[Unit]]`, so we `flatten` it.

Voilà!

Exercise 8: Fixing a bug in `ConcurrentLatch`

There's a bug in our `ConcurrentLatch` implementation—the `decrement` method doesn't handle cancelation properly:

```
def decrement: IO[Unit] =
  state.modify {
    case Outstanding(1, whenDone) =>
      Done() -> whenDone.complete(())
    case Outstanding(n, whenDone) =>
      Outstanding(n - 1, whenDone) -> IO.unit
    case Done() => Done() -> IO.unit
  }.flatten
```

Can you see the problem? (Hint: `modify` can't execute effects itself, so it instead returns an effect.)

How might you approach fixing the problem? (This is *not* easy.)

Solution to Exercise

9.3.2. Using a latch for synchronization

Our use of `Deferred` for the synchronization of the "has there been 13 ticks" state has a small drawback: it pushes the logic of the condition (the `if (count >= 13)` expression) into the `tickingClock` effect. Can we make the clock *unaware* of this condition, but still provide a blocking method of synchronization like `Deferred` gave us?

We can do exactly that with our `CountdownLatch`. There are three interacting concerns:

1. *A shared latch* is initialized with the desired number of ticks, in this case 13.
2. *The beeper* is given the latch and invokes `await` to be blocked until the latch opens.
3. *The ticking clock* is also given the latch and will `decrement` it on every tick.

Example 38. Using `CountdownLatch` *for blocking synchronization. Code available at* `coordination/IsThirteenLatch.scala`.

```
package com.innerproduct.ee.coordination

import cats.effect._
import cats.implicits._
import com.innerproduct.ee.debug._
import scala.concurrent.duration._

object IsThirteenLatch extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    for {
      latch <- CountdownLatch(13)
      _ <- (beeper(latch), tickingClock(latch)).parTupled
    } yield ExitCode.Success

  def beeper(latch: CountdownLatch) =
    for {
      _ <- latch.await
      _ <- IO("BEEP!").debug
    } yield ()

  def tickingClock(latch: CountdownLatch): IO[Unit] =
    for {
      _ <- IO.sleep(1.second)
      _ <- IO(System.currentTimeMillis).debug
      _ <- latch.decrement
      _ <- tickingClock(latch)
    } yield ()
}
```

9.4. Summary

1. We may want to coordinate concurrently executing effects so the behavior of one effect should depend on another. The constraints of that coordination can be described by synchronization: ensuring a relationship between concurrent events (effects).
2. Unless we can synchronize the updates of shared mutable state to be *atomic* (*mutually-exclusive*), modifications of that state can be lost during concurrent actions. In Cats Effect, `Ref` provides atomic updates of a shared value.
3. We may also want to ensure that an effect is evaluated only after some deferred value has been produced (*serialization*). In Cats Effect, `Deferred`

provides write-once, blocking synchronization of a shared value: before the value is available, all readers are blocked. Only when a value has been written are readers unblocked.

4. Blocking means the current execution stops until some (unblocking) condition is triggered. *Semantic blocking* is a term used by Cats Effect where the "logical" execution of the current effect is blocked from further execution, but *not* via the blocking mechanism of the underlying concurrency mechanism, i.e., thread blocking.
5. We can model more complex concurrent behavior by constructing concurrent state machines. They combine atomic updates with blocking synchronization.

[26] Technically our example doesn't have anything to do with transactions, which is part of the domain where this term was first introduced. But the situation is mostly equivalent. You can read more about this and other concurrency control issues at https://en.wikipedia.org/wiki/Concurrency_control.

[27] We wouldn't want to beep-and-stop only if the counter was exactly 13 because, due to timer scheduling issues, our checking may be delayed. If we miss the 13 and check the state when it has incremented to 14, or any other value later, the method would never complete.

[28] Most definitions of sychronization use the term "events" rather than effects, but here we are equating the event with the completion of an effect.

[29] *The Little Book of Semaphores* [6] is an excellent resource to learn more about coordination and synchronization patterns.

[30] Fabio Labella introduced this technique and has popularized it through his talks and public commentary. You can watch his talks and learn more at https://systemfw.org.

Chapter 10. Case study: job scheduler

To put programming with effects into practice, let's design and implement a *job scheduler*—a process that executes user-submitted jobs with its available resources. We'll first talk about jobs, how they are represented and how they changed state. Then we'll manage those jobs so that more than one can run concurrently. As we build up the scheduler's design, we'll also need to answer questions about:

reporting	How is job status tracked, and how do we report it? Could clients register hooks that let them be notified asynchronously?
job control	How can a client cancel a scheduled job?
retry policies	Can we retry a job when it fails? How can we declare different retry policies?
administration	How can we add or remove "workers" available to the scheduler, affecting the overall processing capacity? Can we implement circuit breakers for those workers, so if a particular worker becomes problematic, it can be removed from service?

We'll try to cover all of these, either in-depth, or with suggestions as exercises.

10.1. Jobs

A *job* in this case study will encapsulate some effect we want executed. We'll submit it to the scheduler, and the scheduler will transition the job from state to state. For example, once the job is scheduled, if there's execution capacity the scheduler can run it.

Figure 12. State diagram for a job managed by a scheduler.

We can represent the states of a job as an algebraic data type, `Job`. For now, we'll leave the fields of these states empty.

Example 39. Job state as an algebraic data type.

```
sealed trait Job

object Job {
  case class Scheduled() extends Job
  case class Running() extends Job
  case class Completed() extends Job
}
```

First of all, every `Job` will need an identifier, so let's make a `Job.Id` type.

```
object Job {
  // ...

  case class Id(value: UUID) extends AnyVal

  // ...
}
```

Let's fill in the data our scheduler needs for each of the states:

```
object Job {
  case class Scheduled(id: Id, task: IO[_]) extends Job ①

  case class Running(
      id: Id,
      fiber: Fiber[IO, Either[Throwable, Unit]],  ②
      exitCase: Deferred[IO, ExitCase[Throwable]] ②
  ) extends Job

  case class Completed(id: Id, exitCase: ExitCase[Throwable]) extends Job ③

  // ...
}
```

① A *scheduled* job contains the effect that will be run.

② A *running* job has a `Fiber` representing the executing effect, and a `Deferred` value that will hold the eventual job result as an `ExitCase` value.

③ A *completed* job contains the actual job result as an `ExitCase` value.

You'll notice the `ExitCase` type used in the `Job.Running` and `Job.Completed` cases. It's a data type from Cats Effect that represents if an effect completed successfully, with an error, or was canceled. A simplified definition looks like:

```
sealed trait ExitCase[+E]

object ExitCase {
  case object Completed extends ExitCase[Nothing] ①
  case class Error[+E](error: E) extends ExitCase[E]
  case object Canceled extends ExitCase[Nothing]
}
```

① Note that the `Completed` case doesn't contain the value the effect produced, it only signals a value was successfully produced.

If we fill in our state diagram with the attributes of each state, it now becomes:

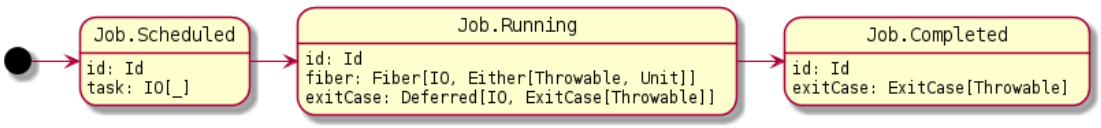

Figure 13. `Job` *as an algebraic data type of possible states.*

We implemented the "nodes" of our diagram—the states. What about the "edges"? They will be methods that transform—as effects—a `Job` from one state to the next. First let's create a scheduled job by generating an id and wrapping the task:

```
object Job {
  // ...

  def create[A](task: IO[A]): IO[Scheduled] =
    IO(Id(UUID.randomUUID())).map(Scheduled(_, task))

  // ...
}
```

Next, to transform a `Job.Scheduled` to a `Job.Running` we need to start the task and capture its eventual completion state as an `ExitCase` value:

```
object Job {
  // ...

  case class Scheduled(id: Id, task: IO[_]) extends Job {
    def start(implicit cs: ContextShift[IO]): IO[Job.Running] =
      for {
        exitCase <- Deferred[IO, ExitCase[Throwable]] ①
        fiber <- task.void
          .guaranteeCase(exitCase.complete) ②
          .start ③
      } yield Job.Running(id, fiber, exitCase) ④
  }

  // ...
}
```

① We create an empty `Deferred` value to hold the eventual `ExitCase` of the task.

② We use the `guaranteeCase` combinator to complete the deferred `ExitCase` value. Like the name implies, the given function is guaranteed to be run no matter what happens during the execution of the task.

③ We `start` the effect, returning a `Fiber` that we keep for tracking purposes.

④ We yield a `Job.Running` value.

Finally, we transform a `Job.Running` to a `Job.Completed`: we await the completion of the `Deferred` exit case value:

```
object Job {
  // ...

  case class Running(
      id: Id,
      fiber: Fiber[IO, Unit],
      exitCase: Deferred[IO, ExitCase[Throwable]]
  ) extends Job {
    val await: IO[Completed] =
      exitCase.get.map(Completed(id, _)) ①
  }

  // ...
}
```

① Semantically block (via the `get` method of the `Deferred`) until the `ExitCase` value has been produced by the running task, then wrap it in a new `Job.Completed` value.

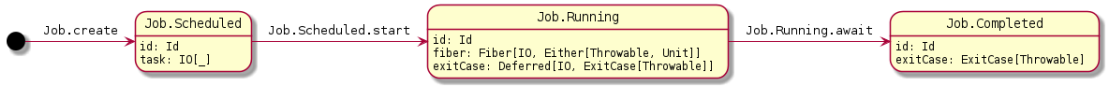

Figure 14. All `Job` *transition functions.*

10.2. Job scheduler

After modeling the behavior of individual jobs, we move on to design our *scheduler*. Its responsibility will be to:

1. accept incoming tasks as scheduled jobs; and
2. execute them, transitioning the jobs through their states.

Looking over the tools and techniques available in Cats Effect, we could imagine a simple scheduler that perhaps uses a `par`-prefixed method like `parSequence` to run multiple jobs in parallel, or maybe we use `Fiber`'s for concurrent tasks. However, those constructs in Cats Effect are opaque. They abstract—for good reasons—how many effects are concurrently running, or are how many are waiting to be run. For this scheduler we need to instead *reify* (make concrete) these concepts so we can "see" what effects are "queued" vs. "executing", and so on.

Let's create an interface for the scheduler's first responsibility, scheduling a task. Once a task is accepted, it will return the identifier of the scheduled job:

```
trait JobScheduler {
  def schedule(task: IO[_]): IO[Job.Id] ①
}
```

① We don't care what the type of value the `task` produces, so we "forget" its type.

For the rest of its responsibilities, the scheduler will manage jobs and their states with a `State` data type:

Example 40. The scheduler's `State` data type.

```
object JobScheduler {
  case class State(
      maxRunning: Int, ①
      scheduled: Chain[Job.Scheduled] = Chain.empty, ②
      running: Map[Job.Id, Job.Running] = Map.empty, ③
      completed: Chain[Job.Completed] = Chain.empty ④
  )

  // ...
}
```

① The maximum number of running jobs.

② The queue of scheduled jobs is modeled as a `cats.data.Chain` to efficently dequeue the first job to run and enqueue newly scheduled jobs at the back of the structure.

③ Running jobs are indexed by their id, for easy lookups.

④ Completed jobs are accumulated once they are complete.

The state itself will be managed within a `Ref` value, to ensure updates are executed atomically.

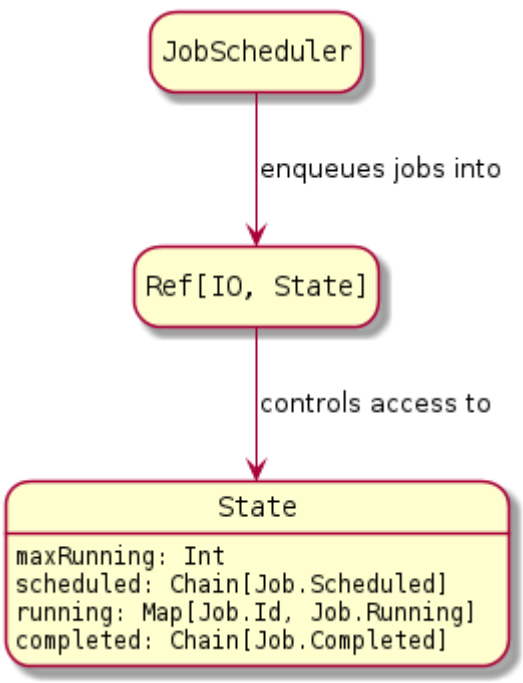

Figure 15. A `JobScheduler` manages state with a `Ref`.

Here's a simple implementation of a `JobScheduler` that manages some `State`:

Example 41. An example implementation of a `JobScheduler` using a `Ref` of `State`.

```
def scheduler(stateRef: Ref[IO, State]): JobScheduler =
  new JobScheduler {
    def schedule(task: IO[_]): IO[Job.Id] =
      for {
        job <- Job.create(task) ①
        _ <- stateRef.update(_.enqueue(job)) ②
      } yield job.id
  }
```

① We create a `Job.Scheduled` value from the `IO` task.

② We enqueue the job into the `State` managed by the `Ref`, where `enqueue` is a method of `State` defined as:

```
def enqueue(job: Job.Scheduled): State =
  copy(scheduled = scheduled :+ job)
```

10.3. Reacting to job state changes

Say we schedule a job—what will subsequently transition the scheduled job to the `Job.Running` state?

```
val jobId: IO[Job.Id] =
  scheduler.schedule(IO("whee").debug) ①
```

① After it is submitted, what will actually run this task?

We need a *reactor*: it continuously waits for job state changes, like a newly scheduled job, and when one happens it reacts:

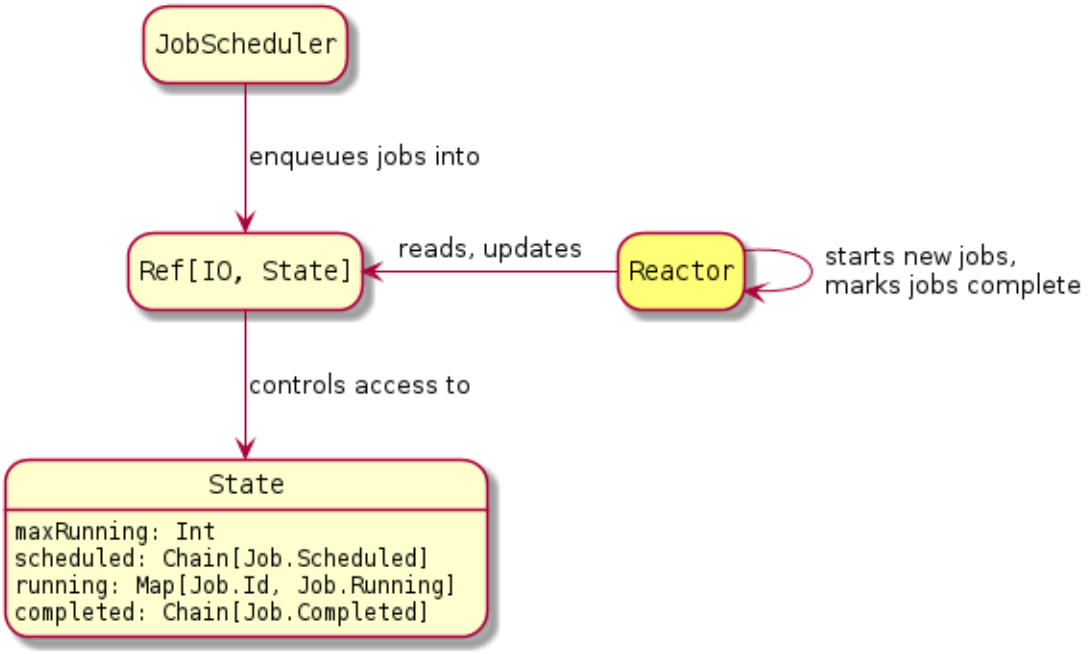

Figure 16. The reactor reacts to events: it manages jobs and tracks their state.

We can model this reactive cycle as a state machine, where the reactor is—abstractly speaking—either *asleep* or *awake*:

when asleep Nothing happens. When a (new) job is scheduled, or a running job completes, we are woken up.

when awake While there is at least one scheduled job, and less than `maxRunning` running jobs:

1. dequeue the next scheduled job;
2. start it;
3. mark it as running; and
4. when the job completes, mark it as completed and ensure the run loop is awoken.

Then, go to sleep.

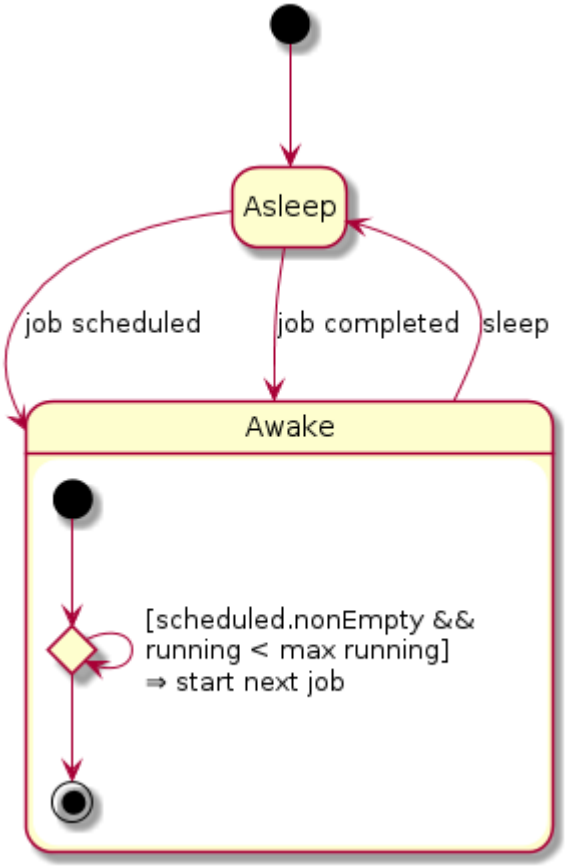

Figure 17. The reactor sleeps until awoken, starts as many jobs as it can, and goes back to sleep.

Concretely, the `Reactor` provides a `whenAwake` effect that acts on the managed scheduler `State`. At the same time, if any job state transitions are taken it notifies the `onStart` and `onComplete` callback effects. We specifically do not manage any sleeping or waking—we'll integrate that in the next section.

Example 42. The `Reactor` interface.

```
trait Reactor {
  def whenAwake(
      onStart: Job.Id => IO[Unit],
      onComplete: (Job.Id, ExitCase[Throwable]) => IO[Unit]
  ): IO[Unit]
}
```

10.3.1. Implementing the reactor

We then implement one using the `Ref[IO, JobScheduler.State]` shared with the `JobScheduler` interface:

Example 43. Implementing the reactor: when awake it starts as many jobs as it can.

```
object Reactor {
  def apply(stateRef: Ref[IO, JobScheduler.State])(
      implicit cs: ContextShift[IO]
  ): Reactor =
    new Reactor {
      def whenAwake(
          onStart: Job.Id => IO[Unit],
          onComplete: (Job.Id, ExitCase[Throwable]) => IO[Unit]
      ): IO[Unit] = {
        startNextJob ①
          .iterateUntil(_.isEmpty) ②
          .void
      }
    }
}
```

① `startNextJob` is defined below. It returns an `Option[Job.Running]` containing the started job, if one was started.

② The `iterateUntil` combinator will run the given effect until the result matches the predicate; we want to stop when no job could be started.

Let's unpack the `startNextJob` effect:

```
def startNextJob: IO[Option[Job.Running]] =
  for {
    job <- stateRef.modify(_.dequeue) ①
    running <- job.traverse(startJob) ②
  } yield running
```

① Modify the shared state, dequeuing a scheduled job if one can be started.

② Start it using `traverse`: `job` is an `Option[Job.Scheduled]`, and `startJob` is `Job.Scheduled ⇒ IO[Job.Running]`, so the result type is `IO[Option[Job.Running]]`.

The `dequeue` method of `State` ensures there are scheduled jobs and more can be started:

```
def dequeue: (State, Option[Job.Scheduled]) =
  if (running.size >= maxRunning) this -> None ①
  else
    scheduled.uncons
      .map {
        case (head, tail) =>
          copy(scheduled = tail) -> Some(head) ②
      }
      .getOrElse(this -> None) ③
```

① Don't start more jobs unless there's capacity.

② The first scheduled job should be started.

③ There are no scheduled jobs to start.

`startJob` transforms the scheduled job to a `Job.Running`, marks it as running once it is started, and ensures that when the job completes, it's marked as completed and the run loop is awoken.

```
def startJob(scheduled: Job.Scheduled): IO[Job.Running] =
  for {
    running <- scheduled.start ①
    _ <- stateRef.update(_.running(running)) ②
    _ <- registerOnComplete(running) ③
    _ <- onStart(running.id).attempt ④
  } yield running
```

① Start the scheduled job, producing a `Job.Running` value. Note that a `Fiber` has been forked and is referenced within the `Job.Running` value.

② `running` is a method of `State` defined as:

```
def running(job: Job.Running): State =
  copy(running = running + (job.id -> job))
```

③ `registerOnComplete` forks an effect to await the completion of the job, then update the state and notify any listeners:

```
def registerOnComplete(job: Job.Running) =
  job.await
    .flatMap(jobCompleted)
    .start

def jobCompleted(job: Job.Completed): IO[Unit] =
  stateRef
    .update(_.onComplete(job))
    .flatTap(_ => onComplete(job.id, job.exitCase).attempt)
```

④ We invoke the `onStart` callback to notify any listeners a job was started.

TODO: compare/contrast this design to using a mutex/semaphore for control; you could just fork effects that race to acquire the mutex to make the next transition(s)

TODO: callback-based implementation for educational purposes; could use more concurrency primitives for event triggering, or something like fs2.

10.3.2. A binary sleeping state machine

Our reactor requires two states, asleep and awake, with effectful transitions between them. Let's extract a state machine that models those states:

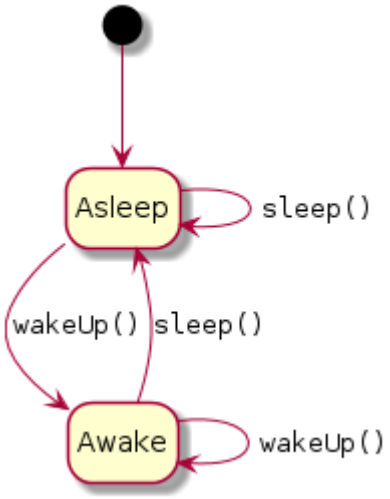

Figure 18. A binary sleeping state machine.

As code, the transition methods form an interface:

```
trait Zzz {
  def sleep: IO[Unit] ①
  def wakeUp: IO[Unit] ②
}
```

① Sleep (semantically block) until `wakeUp` is invoked.

② Wake up (semantically unblock) any sleepers. No effect if already awake.

Exercise 9: Implement `Zzz` as a concurrent state machine

Use the concurrent state machine pattern to implement the `Zzz` interface:

```
object Zzz {
  def apply: IO[Zzz] = ???
}
```

For reference, here is the "recipe" from Section 9.3, "Concurrent state machines":

1. Define an interface whose methods return effects.
2. Implement the interface by building a state machine where:
   a. state (with type `S`) is atomically managed via a `Ref[IO, S]` value;
   b. each interface method is implemented by a state transition function affecting the `Ref`; and
   c. any state-dependent blocking behavior is controlled via `Deferred` values.

10.3.3. Making the reactor sleep and awaken

Now that we have our asleep/awake `Zzz` state machine, we can connect it to the reactor. We need to:

1. Wake up the reactor via `zzz.wakeUp` when a job
   a. is scheduled, or
   b. completes; and
2. The reactor will `zzz.sleep` until woken up, then it will execute `reactor.whenAwake` and go back to sleep.

For the first we extend the behavior of the initial `JobScheduler` implementation to `wakeUp` after a job is scheduled:

```
def scheduler(schedulerState: Ref[IO, State], zzz: Zzz): JobScheduler =
  new JobScheduler {
    def schedule(task: IO[_]): IO[Job.Id] =
      for {
        job <- Job.create(task)
        _ <- schedulerState.update(_.enqueue(job))
        _ <- zzz.wakeUp ①
      } yield job.id
  }
```

① When the job is scheduled, wake up.

Second, we wake up the reactor when a job completes; we use the `onComplete` handler of the `Reactor` itself:

```
def reactor(schedulerState: Ref[IO, State], zzz: Zzz): Reactor =
  new Reactor(
    schedulerState,
    onStart = ???,
    onComplete = (id, exitCase) => zzz.wakeUp ①
  )
```

① When a job completes, wake up.

Finally, for the asleep-awake-asleep transition we create a loop:

```
def loop(zzz: Zzz, reactor: Reactor): IO[Nothing] =
 (zzz.sleep *> reactor.whenAwake(onStart, onComplete)) ①
   .foreverM ②
```

① Sleep until we're awakened, then run the `whenAwake` effect. Repeat this…

② … forever, via the `foreverM` combinator—available to any `Monad`. Since we never exit the loop, the type of value of this effect is `Nothing`.[31]

10.4. Putting everything together

We now have the components that together implement the behavior we want:

1. a `JobScheduler` to enqueue tasks;
2. a `Reactor` to start scheduled jobs and handle completed jobs; and
3. a loop effect that triggers the `Reactor` to wake up, and then go back to sleep.

As a client of these, we can provide these values and effects as a managed resource:

```
object JobScheduler {
  def resource(maxRunning: Int)(
      implicit cs: ContextShift[IO]
  ): IO[Resource[IO, JobScheduler]] =
    for {
      schedulerState <- Ref[IO].of(JobScheduler.State(maxRunning))
      zzz <- Zzz.asleep
      scheduler = new JobScheduler {
        def schedule(task: IO[_]): IO[Job.Id] =
          for {
            job <- Job.create(task)
            _ <- schedulerState.update(_.enqueue(job))
            _ <- zzz.wakeUp
          } yield job.id
      }
      reactor = Reactor(schedulerState)
      onStart = (id: Job.Id) => IO.unit
      onComplete = (id: Job.Id, exitCase: ExitCase[Throwable]) =>
          zzz.wakeUp
      loop =
        (zzz.sleep *> reactor.whenAwake(onStart, onComplete))
          .foreverM
    } yield loop.background.as(scheduler)
}
```

Recall that `background` forks a `Fiber` from an `IO` and manages it as `Resource`, so when the `use` effect of the `Resource` completes, the `Fiber` will be canceled. We don't expose the `loop` to the caller, we instead scope the `JobScheduler` to the lifetime of the "backgrounded" `loop`.

We want this behavior in our application—we use the scheduler as we wish, then everything will get shut down:

```
for {
  resource <- JobScheduler.resource(maxRunning = 2)
  _ <- resource.use { scheduler =>
    ??? ①
  } ②
} yield ???
```

① Use the `scheduler` as much as you want. But…

② … once the `use` effect completes, the (hidden) `loop` will be cancelled, stopping any internal notifications to the `Reactor`.

10.5. Summary

To model jobs, their state and behavior, we used the following techniques:

1. Represent job states as the `Job` algebraic data type.
2. Transition between job states via effectful methods, e.g., `Job.Scheduled.start(): IO[Job.Running]`.
3. For a running job, we
   a. model the executing effect as a `Fiber` (Section 4.2, "Gaining control with `Fiber`"); and
   b. model the use eventual `ExitCase` result as a `Deferred` value (Section 9.2, "Write-once synchronization with `Deferred`").

The job scheduler:

1. Manages its `JobScheduler.State` value with a `Ref`.
2. Awakens the `Reactor` via the `Zzz.wakeUp` effect.

The reactor responds to events by:

1. Managing state with a `Ref`.
2. Forking effects that await the completion of jobs.

To link these components together:

1. The `Zzz` coordination behavior is implemented as a concurrent state machine.
2. We manage the lifetime of the reactive loop with a `Resource`, but only expose the `JobScheduler` value to the user of it.

[31] Since there is no value of type `Nothing`, we could never produce one, which implies this effect can't ever finish.

Chapter 11. Conclusion

> I privately say to you, old friend... please accept from me this unpretentious bouquet of early-blooming parentheses: ((((())))).
>
> — J.D. Salinger, *Raise High the Roof Beam, Carpenters & Seymour: An Introduction*

You did it! Woo!

To finish, let's summarize the most important concepts of the book, and how they are represented in Cats Effect.

Effects help us reason about our programs.

An "effect" is what happens when a computation is executed. We name these effects by giving them a type—it tells us what kind of effects the program will perform, in addition to the type of the value it will produce. At the same time, effect types separate describing what we want to happen from actually making them happen. We can safely substitute the description of effects up until the point we run them.

Cats Effect `IO` lets us safely represent side effects.

The `IO` effect type encapsulates any kind of computation, even (most-importantly!) side effects. We can safely compose `IO` values to produce new effects, and only when we run them (via `unsafe`-prefixed methods) does anything happen.

capture side effects	`IO.delay`, ...
effect composition	`map`, `mapN`, `flatMap`, ...
execute effects	`unsafeRunSync`, ...
effect-based applications	`IOApp`

Parallelism lets us perform independent work more efficiently.

Parallelism is implemented by scheduling work onto multiple threads, so if there are `n` threads then at most `n` effects may be run in parallel. Parallel versions of more familiar (but sequential) methods are prefixed with `par`: `parMapN`, `parTraverse`, and so on. If an error occurs during parallel execution, the remaining effects are cancelled.

parallel execution	`parMap`, `parTraverse`, `parSequence`, `parTupled`

Concurrency allows us to declare certain execution behavior without explicitly specifying how it is scheduled.

With Cats Effect, we can `start` a concurrent effect and use the returned `Fiber` to wait until it completes (`join`) or request it to stop (`cancel`). At a higher level, concurrently "racing" two effects lets us subsequently act with the knowledge of which one finished first.

fork an effect	`start`
concurrent control	`join`, `cancel`
race multiple effects	`IO.race`, `IO.racePair`

Contexts represent a pool of computational resources "where" computations can be executed.

When we perform parallel or concurrent effects they get scheduled onto the available threads in the current context. But we can control the execution of effects via those contexts, too. Within a context, an asynchronous boundary will reschedule the subsequent execution, so that other concurrently running effects can have an opportunity to progress. At the same time, having multiple contexts keeps executing effects isolated from each other. Cats Effect supports the common application pattern where non-blocking, CPU-bound work is in one context, and blocking, I/O-bound work is in another.

yield execution / reschedule	(CE2) `IO.shift` (CE3) `IO.cede`, `evalOn`
declare blocking effects	(CE2) `Blocker` (CE3) `IO.blocking`

Asynchrony allows us to decouple a computation from the code that handles its result—the *continuation*.

Instead of waiting for the result of a computation, we can defer that result handling so the current effect can continue without being blocked. In Cats Effect we can then build an effect *from* an asynchronous computation, allowing us to integrate with asynchronous types like `Future`.

integrate asynchrony	`IO.async`, `IO.fromFuture`

Resources separate the acquisition and release of state from its use.

We can explicitly manage the lifecycle of state by declaring a `Resource`, and then compose multiple resources together. Resources can be composed serially or in

parallel, and the composed resource will ensure that its constituent resources are acquired and released in the proper order. Resources are a natural data type for modeling dependencies whose lifecycle must be managed.

create resources	`Resource.make`
resource composition	`map`, `mapN`, `flatMap`, etc., plus `par`-prefixed versions
use resources	`use`

Testing effects requires their execution.

This can be a problem when those effects actually "launch the missiles", whereas we may want to only assert that such side effects *will* happen. To mitigate the danger we can create an interface that abstracts over those effectful methods so that we can provide fake implementations when testing. At the level of effect scheduling, Cats Effect's `TestContext` lets you replace the real execution system with one that the programmer can control.

control effect scheduling in tests	`TestContext`

Coordinating concurrent effects requires abstractions that represent (and respect) synchronization constraints.

For example, we can use a `Ref` to share "mutable" state, and it ensures updates are properly synchronized. Or we can use `Deferred` when we want to (semantically) block one effect until another effect produces a value. With these two primitives we can then build more complex concurrent behaviors with techniques like concurrent state machines.

atomically update shared state	`Ref`
blocking synchronization	`Deferred`

11.1. Next steps

This book is part of a larger curriculum and community. Here are some suggested steps for what to explore next:

Ask questions, get advice and meet members of the Cats Effect community

Join the Cats Effect chatroom.[32]

Build and use more powerful abstractions atop Cats Effect

For example, fs2[33] is a powerful library that lets you build sophisticated flow control structures, doobie[34] wraps JDBC database operations, and http4s[35] is an

entire HTTP stack. Other libraries and projects that use Cats Effect are listed on the main Cats Effect site.[36]

Learn more about functional programming concepts and patterns

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

Learn about architecting an application with functional programming, including Cats Effect

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

Learn how to build a functional domain model and build composable abstractions

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

[32] https://gitter.im/typelevel/cats-effect

[33] https://fs2.io

[34] http://tpolecat.github.io/doobie/

[35] http://http4s.org/

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

Glossary

algebraic data type

An algebraic data type (ADT) is a data type built out of combinations of AND's and OR's of other types. In Scala 2, AND's are encoded using a `case class`, and OR's are encoded as a `sealed trait`. The book *Essential Scala* [2] talks about ADTs in great depth and is an excellent resource for learning more.

applicative (functor)

A functor that can transform multiple structures, not just one. In Cats, the canonical applicative method is `mapN`.

asynchronous

An asynchronous computation is started without the caller waiting for the result to be produced. Once the result is computed, it can be handled by a provided continuation (or callback). If no result is eventually required, the asynchronous computation is often termed *fire-and-forget*.

asynchronous boundary

A syntactic boundary between the current computation and its asynchronous continuation. For example, an effect that represents the runtime rescheduling the remainder of the computation, possibly resuming it on another thread.

atomic

Two concurrent effects are atomic when they cannot happen at the same time, i.e., they are mutually-exclusive.

blocking

Waiting to consume an asynchronously produced result.

call-by-value

A strict evaluation strategy for calling functions or methods: arguments are evaluated before the function or method is called.

call-by-name

A lazy evaluation strategy for calling functions or methods: sub-expressions are evaluated when the expression is referenced by the function or method.

cancelation boundary

If encountered as an effect executes, the cancelation status for the current effect is checked, and if that effect has been canceled then execution will stop.

concurrent

Computations are concurrent when their execution *lifetimes* overlap. Concurrency emphasizes the non-deterministic and structural aspects of computation: we can't tell when anything happens, only that their lifetimes overlap. Antonym: *sequential*.

continuation

Code that will be executed "next". When viewed as part of a larger sequence of steps, the continuation represents an unexecuted portion of the whole. For example: callbacks; the resumption of an effect after an asynchronous boundary.

bracketed effect

A composite effect with explicit pre- and post-effect steps. It is guaranteed that if the pre-effect succeeds, then the post-effect will always be executed, even if the "bracketed" effect itself has failed.

effect

What happens when a computation is executed. We can encapsulate different effects as types, which helps us identify the effect along with the type of value that the effect produces.

evaluation

The process of transforming an expression into a value. There are multiple evaluation strategies: call-by-value, call-by-name, etc.

expression

Expressions can be evaluated into values. Every expression has a type.

finalizer

An effect that will "clean up" (*finalize*) any state when it is no longer needed. A generalization of Java's `finally` clause of a `try`/`catch` block. Finalizers are often paired with state allocation effects to form a bracketed effect or resource.

functor

Something you can `map` over, changing its "contents" (or output) but not the structure itself.

higher-order function

A function which takes a function as an input, and/or produces a function as an output.

local reasoning

The ability to understand a definition of a function or value because it doesn't

depend on anything except its explicit inputs. All information is *local* to the definition.

monad

A monad is a mechanism for sequencing computations: this computation happens after that computation. Typically defined by the `flatMap` method on a type.

mutual-exclusion

Two concurrent effects are mutually-exclusive when only one can happen at the same time, i.e., atomic.

parallel

Computations are parallel when their executions occur at the same *instant* in time. Its main concern is the operational utilization (efficiency) of resources used during the execution. Parallelism requires determinism: no matter how many resources you have, you *must* produce the same answer. Antonym: *serial*.

referential-transparency

An expression is referentially-transparent when the meaning of an expression doesn't change if it is substituted with the value it evaluates to.

resource

A managed value, defined by two effects, acquisition (producing the value) and release ("cleaning up" the value). It can be subsequently used so that the acquired value is guaranteed to be released.

safe

Respects substitution; is referentially transparent.

semantic blocking

The "logical" execution of the current effect is blocked from further execution, but *not* via the blocking mechanism of the underlying concurrency mechanism, i.e., thread blocking.

serialization

Two concurrent effects are serialized if one must happen after the other.

side effect

A change in the environment that causes substitution to become non-deterministic: the expression and its value will no longer have the same meaning.

substitution

The process of replacing an expression with its definition.

synchronization

ensuring a particular relationship between concurrent events; for example, serialization, mutual-exclusion.

thunk

A delayed computation that may optionally memoize its result. In Scala, often represented as a `lazy val` or a call-by-name parameter.

type

A syntactic method to label the constructs of a program. For example, expressions and values both have a type.

unsafe

Performs a side effect; does **not** respect substitution; is **not** referentially transparent.

value

Values are immutable; every value has a type.

Appendix A: Cheatsheets

A.1. Cats typeclasses and extension methods

For more details, see the *Cats typeclasses documentation.*[37]

Extension methods based on Cats typeclasses are usually imported like:

```
import cats.implicits._
```

Table 1. Common extension methods provided by the `Functor[F[_]]` *typeclass on a* `F[A]` *value.*

Extension Method	Example
`def map[B](f: A => B): F[B]`	`val fb: F[B] = fa.map(a => f(a))`
`def as[B](newValue: => B): F[B]`	`- fa.map(_ => b)` `+ fa.as(b)`
`def void(): F[Unit]`	`- fa.map(_ => ())` `+ fa.void`

Table 2. Common extension methods provided by the `Applicative[F[_]]` *typeclass on a* `F[A]` *value.*

Extension Method	Example
`mapN`	`val fc: F[C] = (fa, fb).mapN((a, b) => c)`
`tupled`	`- (fa, fb).mapN((a, b) => (a, b))` `+ (fa, fb).tupled`
`*>`	`- (fa, fb).mapN((_, b) => b)` `+ fa *> fb`

Extension Method	Example
`<*`	`- (fa, fb).mapN((a, _) => a)` `+ fa <* fb`
`replicateA`	`- List.fill(3)(fa).sequence` `+ fa.replicateA(3)`

Table 3. Common extension methods provided by the `Monad[F[_]]` typeclass on a `F[A]` value.

Extension Method	Example
`flatMap`	`val fb: F[B] = fa.flatMap(a => f(a))`
`flatTap`	`- a.flatMap(a => f(a).as(a))` `+ a.flatTap(f)`
`>>`	`- a.flatMap(_ => fb)` `+ a >> b`

Table 4. Common extension methods provided by the `ApplicativeError[F[_], E]` typeclass on a `F[A]` value.

Extension Method

`def attempt: F[Either[E, A]]`
 	
`def adaptError(pf: PartialFunction[E, E]): F[A]`
 	
`def handleError(f: E => A): F[A]`
 	
`def handleErrorWith(f: E => F[A]): F[A]`
 	
`def onError(pf: PartialFunction[E, F[Unit]]): F[A]`

```
def recover(pf: PartialFunction[E, A]): F[A]

def recoverWith(pf: PartialFunction[E, F[A]]): F[A]

def redeem[B](recover: E => B, f: A => B): F[B]
```

Table 5. Common methods provided by the `MonadError[F[_], E]` *typeclass.*

Method
`def rethrow[A, E](fa: F[Either[E, A]]): F[A]`

TODO: Traverse

TODO: Parallel: parMapN, parTupled, parTraverse, parSequence

A.2. Cats Effect data types

A.2.1. Clock

TODO

A.2.2. ContextShift

TODO

A.2.3. Deferred

TODO

A.2.4. Fiber

TODO

A.2.5. IO

Table 6. Common factory methods of `IO`.

Signature	Use
`val cancelBoundary: IO[Unit]`	"Returns a cancelable boundary — an `IO` task that checks for the cancellation status of the run-loop and does not allow for the bind continuation to keep executing in case cancellation happened."
`def delay[A](body: => A): IO[A]`	Delay a side effect that produces a value of type `A`.
`val never: IO[Nothing]`	An effect that never produces a value.
`def pure[A](a: A): IO[A]`	Construct an effect from an existing "pure" value.
`def race[A, B](lh: IO[A], rh: IO[B])(implicit cs: ContextShift[IO]): IO[Either[A, B]]`	Run two effects concurrently, returning the result of the winner in an `Either`. The "loser" of the race will be cancelled.
`def raiseError[A](e: Throwable): IO[A]`	Lift an exception into an effect.
`def sleep(duration: FiniteDuration)(implicit timer: Timer[IO]): IO[Unit]`	(Semantically) block for the given duration.
`val unit: IO[Unit]`	An effect which does nothing. An alias for `IO.pure(())`.

A.2.6. Ref

TODO

A.2.7. Resource

Table 7. Common methods of a `Resource[IO, A]` value.

Method	Signature
use	`def use[B](f: A => IO[B]): IO[B]`

A.2.8. `Timer`

TODO

[37] https://typelevel.org/cats/typeclasses.html

Appendix B: Abstracting effects with typeclasses

While this book focuses on the concrete `cats.effect.IO` data type, the design of the Cats Effect library separates effectful behaviors—parallelism, concurrent control, and so on—using typeclasses. Those typeclasses are parameterized by an effect type.[38] The library then provides typeclass instances for the concrete `cats.effect.IO` data type.

Example 44. Concrete vs. abstract effect types.

`IO` *is a concrete type.*

```
def doSomething(): IO[Int]
```

`F[_]` *is an abstract type parameter that can be used to specify argument and return types.*

```
def doSomething[F[_]]: F[Int]
```

Why might this matter? When we write programs with a concrete type like `IO`, we are being very explicit about our choice of effect type. `IO` is an effect that can "do anything, even side effects". It is *very* powerful. But our programs might not be using all of that power in every part. Consider this method:

```
def combineStuff(e1: IO[Int], e2: IO[Int]): IO[Int] =
  for {
    i1 <- e1
    i2 <- e2
  } yield i1 + i2
```

Is what we're doing really dependent upon the `IO` effect type? Are we doing something like invoking `start` to create a new `Fiber`, or using `guarantee` to ensure some extra effect happens? No! In this case our code only relies on the ability to `flatMap`, in the form of a `for`-comprehension. So we could refactor our method like so:

```
import cats._ ①
import cats.implicits._ ②

def combineStuff[F[_]: Monad](e1: F[Int], e2: F[Int]): F[Int] = ③
  for {
    i1 <- e1
    i2 <- e2
  } yield i1 + i2
```

① We import from `cats` the `Monad` typeclass which encapsulates the ability to `flatMap` an effect type.

② This import provides extension methods to access `flatMap` and other methods defined in the typeclass.

③ We require an effect type as the type parameter `F[_]`, along with a `Monad[F]` typeclass instance (dependency) to be available. (The `F[_]: Monad` syntax is called a *context bound*.) The method arguments now have type `F[Int]`.

You could argue that we've made the method signature more complex, and you'd be right! We replaced a single `IO` type with both a type parameter and context bound. But we've also gained two important benefits:

1. *Effect polymorphism*

   It is now the *caller* of `combineStuff` who decides what the effect type will be, not the definition of `combineStuff` itself. `combineStuff` may be called with `IO` arguments, or `Future` or `Option` or indeed any `Monad`. Our method can handle them all, without having to care what their concrete type is.

2. *Error-reduction*

   The possible implementations of `combineStuff` is severely restricted because the types of the arguments are abstract: we only know that `e1` and `e2` are of type `F`, and the only methods we can invoke are those provided by the `Monad[F]` typeclass instance. This reduces the possibilities of errors, because we can't do *anything* except what the constraints (typeclass instances) allow. There is literally less that can go wrong.[39]

These benefits—polymorphism and error reduction—accrue during any refactoring from a concrete type to an abstract one. So if your fellow programmers are comfortable with the existing `IO` type, type constructors and typeclasses, consider taking advantage of the Cats Effect typeclasses in your codebase.

Here's a brief overview of the typeclasses Cats Effect 2 defines:[40]

`Bracket[F[_]]`	"Can safely acquire and release resources."
`Sync[F[_]]`	"Can suspend (describe) synchronous side effecting code in `F`."
`Async[F[_]]`	"Can suspend synchronous/asynchronous side effecting code in `F`."
`Concurrent[F[_]]`	"Can concurrently start or cancel the side effecting code in `F`."
`Effect[F[_]]`	"Allows lazy and potentially asynchronous evaluation of side effecting code in `F`."
`ConcurrentEffect[F[_]]`	"Allows cancelable and concurrent evaluation of side effecting code in `F`."

Remember, `cats.effect.IO` implements instances for every one these typeclasses! For more details about them see the Cats Effect typeclasses documentation.

[38] This is sometimes referred to as *effect polymorphism* when the effect type is an abstract type on an interface.

[39] The task of implementation itself also becomes simpler, in the sense that there are fewer choices of methods one can invoke. We assert that it is often the case that there is only one method available for a given (sub-)task.

[40] These set of typeclasses *will* change in Cats Effect 3, but hopefully in a way that makes them simpler and more usable.

Appendix C: Changes in Cats Effect 3

While Cats Effect 3 significantly changes the base set of typeclasses that specify its overall behavior, the amount of differences with respect to the concepts and code in *Essential Effects* is fairly small.

C.1. Method changes

- `IO.async` has been renamed to `IO.async_`, and a new `IO.async` method allows the asynchronous process to return a cancelation token rather than `Unit`.

```
- IO.async { cb => ??? }
+ IO.async_ { cb => ??? }
```

- `IO.start` no longer takes an implicit `ContextShift`.
- `Fiber.join` now returns `Outcome`, where an `Outcome` is one of `Succeeded`, `Errored`, or `Cancelled`. You can use `Outcome.embed(onCancel: F[A])` or `Output.embedNever` to get an `F[A]`.

```
- a <- fiber.join
+ outcome <- fiber.join
+ a <- outcome.embed(onCancel = ???)
```

- `IO.shift` is renamed to `IO.cede`; both allow you to specify an asynchronous boundary so that the current effect is rescheduled.

```
- IO.shift
+ IO.cede
```

  Additionally, in CE2 there is `IO.shift(ec: ExecutionContext)`, which now becomes a *member* method `evalOn`:

```
- IO.shift(ec) *> someEffect
+ someEffect.evalOn(ec)
```

- `IO.cancelBoundary` is removed. Cancelation boundaries are now automatically inferred from `flatMap` calls.
- `Deferred.complete` now returns `IO[Boolean]` vs. `IO[Unit]`.

C.2. Data type changes

- `Fiber[F[_], A]` is now `Fiber[F[_], E, A]`, and `FiberIO[A]` is an alias for `Fiber[IO, Throwable, A]`.
- `ContextShift` has been removed. If you want an effect to execute on a different context, use `evalOn(ec: ExecutionContext)` directly.

```
  val ec = ???
- val pool: ContextShift = ???
- pool.evalOn(ec)(IO("on pool").debug)
+ IO("on pool").debug.evalOn(ec)
```

- `Blocker` has been removed. Use `IO.blocking` to construct blocking effects directly.

```
- Blocker[IO].use { blocker =>
-   blocker.blockOn(IO("on blocker").debug)
- }
+ IO.blocker(IO("on blocker").debug)
```

C.3. Package changes

- `Ref` and `Deferred` in the `cats.effect.concurrent` package are moved to package `cats.effect.kernel` and will be aliased to the top-level `cats.effect` package.
- Other concurrency data types like `Semaphore` have been moved from package `cats.effect.concurrent` to `cats.effect.std`.
- `TestContext` has been moved from module `cats-effect-laws`, package `cats.effect.utils`, to module `cats-effect-testkit` with package `cats.effect.testkit`.

```
- "org.typelevel" %% "cats-effect-laws" % CatsEffect2Version % Test
+ "org.typelevel" %% "cats-effect-testkit" % CatsEffect3Version % Test
```

Appendix D: Solutions to selected exercises

Some solutions are presented directly after the exercises in the book for pedagogial purposes. Those that do not have their solutions described here.

D.1. Effects: evaluation and execution

Solution to Exercise 1: Timing

```
package com.innerproduct.ee.effects

import java.util.concurrent.TimeUnit
import scala.concurrent.duration.FiniteDuration

object Timing extends App {

  val clock: MyIO[Long] =
    MyIO(() => System.currentTimeMillis) ①

  def time[A](action: MyIO[A]): MyIO[(FiniteDuration, A)] =
    for { ②
      start <- clock
      a <- action
      end <- clock
    } yield (FiniteDuration(end - start, TimeUnit.MILLISECONDS), a)

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

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

① We capture the current time, which doesn't respect substitution, within a `MyIO`.

② We use a `for`-comprehension to sequence multiple effects: first we capture the start time, then perform our effect, then capture the end time in order to calcuate the duration of the action.

Running the `Timing` program produces:

```
hello!
'hello' took 66 milliseconds
```

D.2. Cats Effect `IO`

Solution to Exercise 2: Ticking Clock

```
package com.innerproduct.ee.io

import cats.effect._
import cats.implicits._
import scala.concurrent.duration._

object TickingClock extends IOApp {

  def run(args: List[String]): IO[ExitCode] =
    tickingClock.as(ExitCode.Success)

  val tickingClock: IO[Unit] =
    for {
      _ <- IO(println(System.currentTimeMillis)) ①
      _ <- IO.sleep(1.second) ②
      _ <- tickingClock ③
    } yield ()
}
```

① We first print the current time via `System.currentTimeMillis`, then

② sleep for one second, and then

③ use recursion to do it again.

Running the app you'll see something like:

```
1598395213465
1598395214498
1598395215503
1598395216508
```

...and so on.

D.3. Parallel execution

D.4. Concurrent Control

D.5. Shifting Contexts

D.6. Integrating asynchrony

Solution to Exercise 4: `java.util.concurrent.CompletableFuture`

Our task is to implement the handler we pass to the `CompletableFuture`. Its job is to delegate the result to the callback `cb`.

The type signature of the handler reflects its Java origin: its input parameter is a tuple of the successful value and a possible exception, so we can assume only one part of the tuple will be "filled in" and the other part will be `null`.

```
- val handler: (A, Throwable) => Unit = ???
+ val handler: (A, Throwable) => Unit = {
+   case (a, null) => cb(Right(a)) ①
+   case (null, t) => cb(Left(t))  ①
+   case (a, t) => sys.error(s"CompletableFuture handler should always have one null, got: $a,
$t")
+ } ②
```

① If the `CompletableFuture` successfully produces an `A`, we can expect the `Throwable` to be null, and vice versa if there was an error.

② If both are `null`, that would be a programming error in the JDK, so we throw an error.

Solution to Exercise 5: Never!

`IO.async` takes a function that provides a callback to report the asynchronous result. So if we never want to return a result, we simply don't invoke the callback!

```
val never: IO[Nothing] =
- IO.async(???)
+ IO.async(cb => ()) ①
```

① We return a `Unit` to conform to the signature of `IO.async`.

We've essentially said "thanks for the callback, but no thanks. Here's a `Unit` to acknowledge you gave me the callback (that I'm not going to use)."

Solution to Exercise 6: Why does `IO.fromFuture` require a `Future` inside an `IO`?

We require the `Future` to be inside an `IO` because creating a `Future` has a side effect: it schedules the `Future` to be executed. In the world of effects we need to delay any side effects, so the creation of the `Future` needs to be wrapped.

D.7. Managing Resources

Solution to Exercise 7: Early-release of Resources

When we compose a `Resource` with `flatMap` or `for`-comprehension, we're nesting the next resource (the `Config`) "inside" the first one (the `Source`). To avoid this we want to instead `use` the `Source` resource immediately to produce the `Config`, and directly lift this effect back into a `Resource`:

```
lazy val configResource: Resource[IO, Config] =
- for {
-   source <- sourceResource
-   config <- Resource.liftF(Config.fromSource(source))
- } yield config
+ Resource.liftF(sourceResource.use(Config.fromSource))
```

After making this change we can see the configuration `Source` gets closed once it isn't needed anymore:

```
[ioapp-compute-0] > opening Source to config
[ioapp-compute-0] read Config(exampleConnectURL)
[ioapp-compute-0] < closing Source to config
[ioapp-compute-0] > opening Connection to exampleConnectURL
[ioapp-compute-0] (results for SQL "SELECT * FROM users WHERE id = 12")
[ioapp-compute-0] < closing Connection to exampleConnectURL
```

D.8. Testing Effects

D.9. Concurrent Coordination

Solution to Exercise 8: Fixing a bug in `ConcurrentLatch`

`decrement` is a composed effect; it first executes `modify`, and then returns another effect which is subsequently executed via `flatten`. If the "outer" `decrement` is cancelled...

```
def decrement: IO[Unit] =
  state.modify {
    case Outstanding(1, whenDone) =>
      Done() -> whenDone.complete(())
    case Outstanding(n, whenDone) =>
      Outstanding(n - 1, whenDone) -> IO.unit
    case Done() => Done() -> IO.unit
  }.flatten ①
```

① ... after the "inner" `modify` completes, but before `flatten` does, then the state will be `Done` but `whenDone` will never have been completed, so any previously `await` calls will never be unblocked.

To solve this problem requires the ability to mark effects as *uncancelable*. This is a powerful ability, so use it with caution. For our case, we mark the composed effect, so the `modify` and `flatten` will always execute together:

```
def decrement: IO[Unit] =
  state.modify {
    case Outstanding(1, whenDone) =>
      Done() -> whenDone.complete(())
    case Outstanding(n, whenDone) =>
      Outstanding(n - 1, whenDone) -> IO.unit
    case Done() => Done() -> IO.unit
  }
  .flatten
  .uncancelable ①
```

① Mark the composed effect as uncancelable.

References

- [1] Dave Gurnell and Noel Welsh. https://creativescala.com
- [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