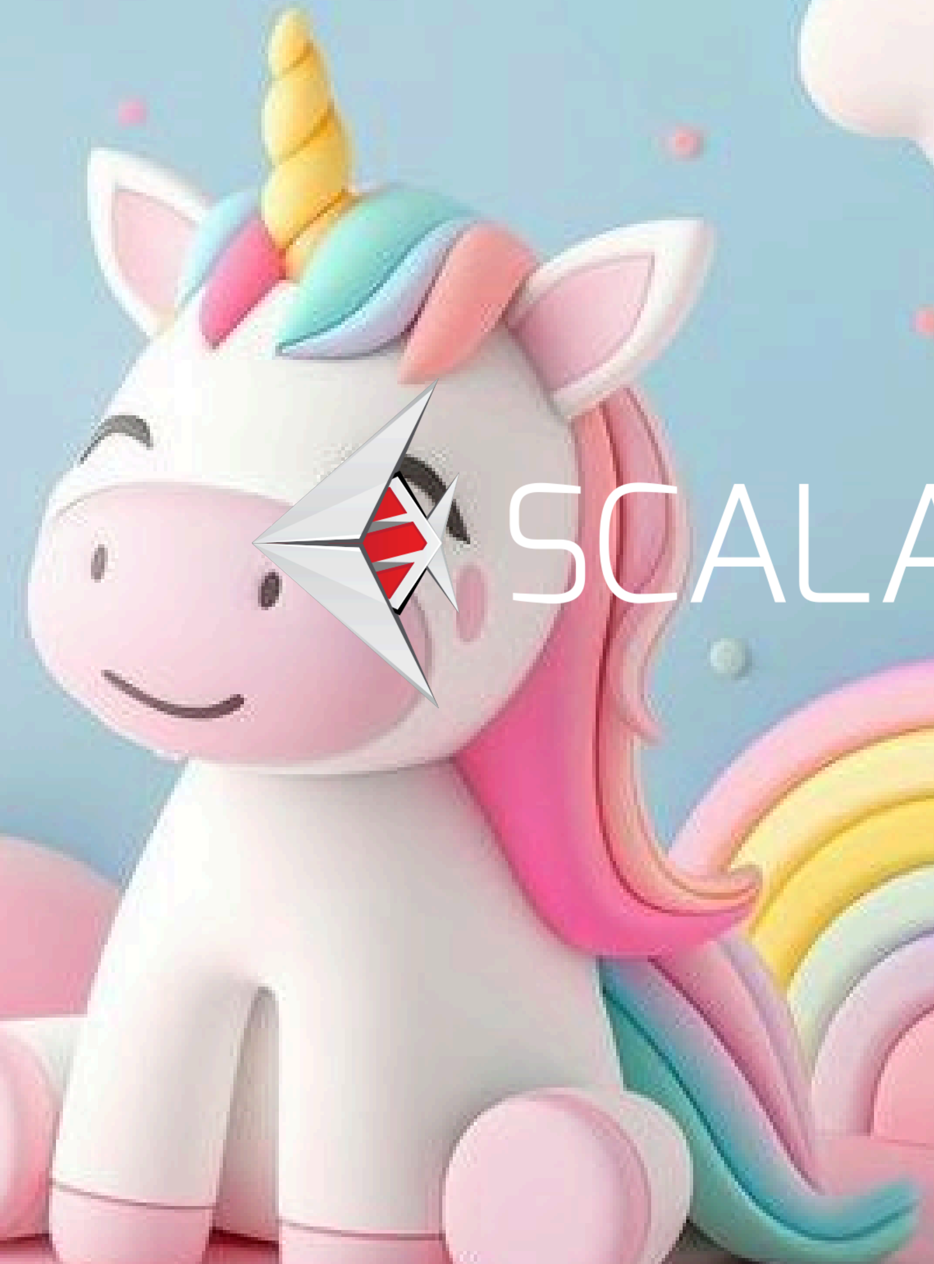
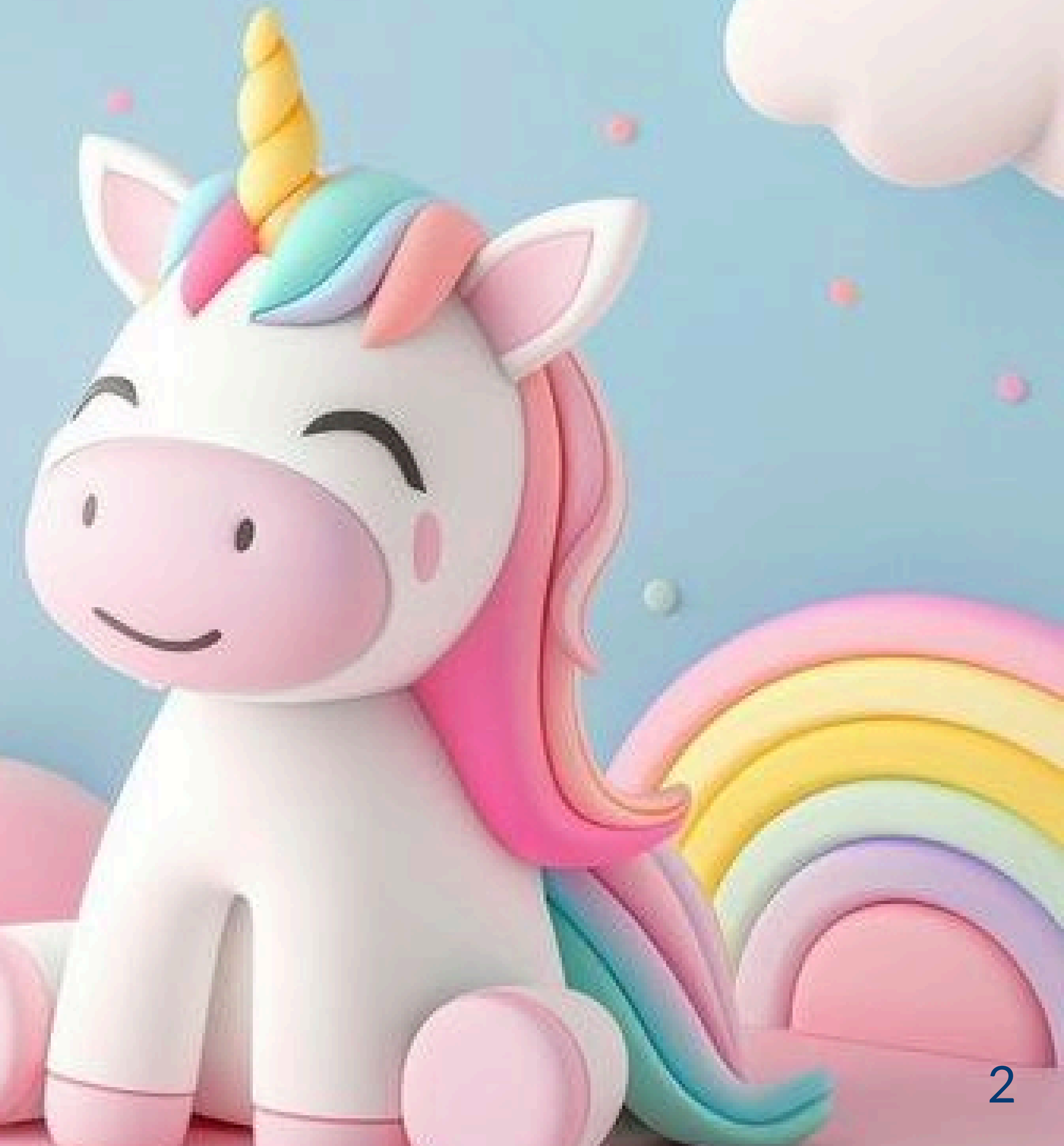
# Yo Dawg, Heard You Want to Flatmap Your Direct-style

Effect System in Scala Using Capability Passing Style



# Agenda

- 👋 Who Am I?
- ❤️ Effects and 💔 Side Effects
- 🍱 Scala Monadic Effect Systems
- 🛠️ Build Your Own Effects System
- ➕ Adding Monadic Operations
- 🏁 Conclusions and References



# Who Am I?

- Hello there 👋, I'm **Riccardo Cardin**,
  - An Enthusiastic Scala Lover since 2011 💯

# Effects and Side Effects

# Why We ❤ Functional Programming

- We have the **substitution model** for reasoning about programs

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

- The substitution model enables **local reasoning** and **referential transparency**
  - We don't need to look at the implementation
  - Original program and the substituted program are *equivalent*
- We call these functions *pure* functions

# We Live in an Imperfect World 💔

> Model a coin toss, but with a twist: the gambler might be too drunk and lose the coin

```scala
import scala.util.Random

def drunkFlip(): String = {
  val caught = Random.nextBoolean()
  val heads =
    if (caught) Random.nextBoolean()
    else throw new Exception("We dropped the coin")
  if (heads) "Heads" else "Tails"
}
```

# We Live in an Imperfect World 💔

- We can't use the substitution model for all programs
  - If the `drunkFlip` function throws an *exception*, the substitution model breaks
- Programs that interact with a context outside the function
  - The result of the `drunkFlip` function depends on the state of the world
- Multiple calls to `drunkFlip` can return different results

# Side Effects

- **Side Effect**: An *unpredictable change* in the state of the world
  - *Unmanaged*, they just happen

```
// What happens if b is equal to zero?
def divide(a: Int, b: Int): Int = a / b
```

- We call `divide` an *impure* function
- The best we can do is to *track* and push them to the *boundaries* of our system

# The Effect Pattern

When a side effect is *tracked* and *controlled* we call it an **effect**

1. The *type* of the function should tell us what effects it can perform and what's the type of the result
   - The `drunkFlip` deals with *non-determinism* and *errors*
2. We must separate the *description* from making it happen
   - We want a *recipe* of the program.
   - **Deferred execution** & **referential transparency**

The pattern lets us use the **substitution model** again 🚀 🎉

# An Effect Example

Effects have the form of a generic type `F[A]`

- The `Option[A]` type models the conditional lack of a value

```scala
val maybeInt: Option[Int] = Some(42)
val maybeString: Option[String] = maybeInt.map(_.toString)
```

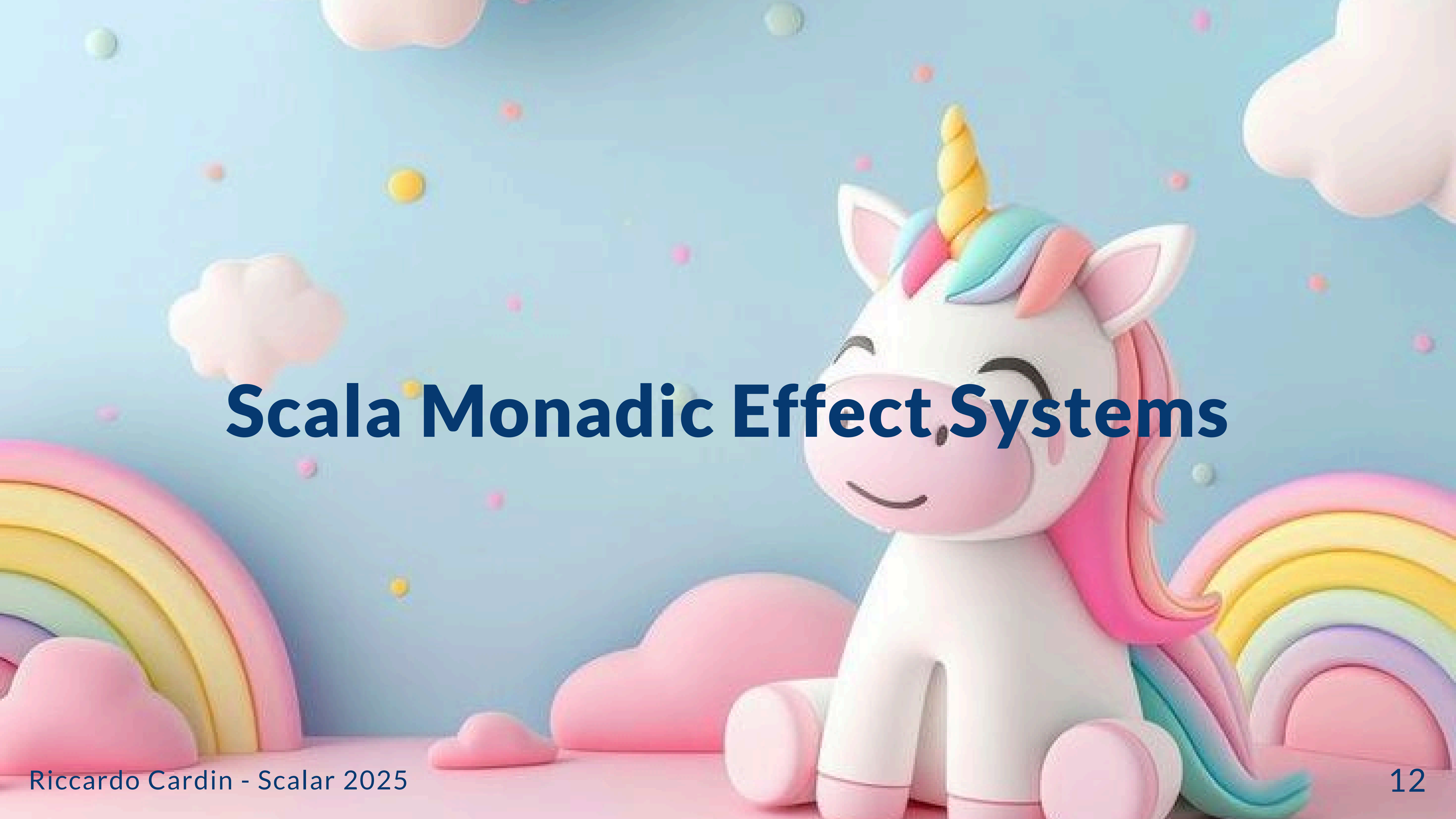
- Composing function returning effects is not trivial
  - `F[_]` must be a *monad* so we can use `flatMap` and `map`
  - Different monads are *hard to compose* (Monad Transformers)

# Effect Systems

An **Effect System** is the implementation of the *Effect Pattern*

- It puts side effects in a *box*
- It replaces side-effecting operations in standard libraries
- It provides structures to manage effects

In an effect system, a side effect 👎 becomes an effect 👍



# Scala Monadic Effect Systems

# Cats Effect

- Cats Effect uses the `IO[A]` data type to model effects
  - `IO[A]` is an *über effect* that models any effectful computation that returns a value of type `A` and can fail with a `Throwable`
  - It's a *monad* so we must use `flatMap` and `map` to compose effectful functions
  - `IO[A]` is **referentially transparent** and **lazy**
  - Redefines the effectful part of the Standard Library
  - Implements *structured concurrency*

# Cats Effect

Let's rewrite the `drunkFlip` function using the `IO` effect

```
def drunkFlip: IO[String] =
  for {
    random <- Random.scalaUtilRandom[IO]
    caught <- random.nextBoolean
    heads <-
      if (caught) random.nextBoolean
      else IO.raiseError(RuntimeException("We dropped the coin"))
  } yield if (heads) "Heads" else "Tails"
```

The `drunkFlip` function returns a *recipe* of the program

# Cats Effect

The library provides many ways to *run* the effect

```scala
object Main extends IOApp.Simple {
  def run: IO[Unit] = drunkFlip.flatMap(result => IO.println(result))
}
```

There are also some *unsafe* methods to run the effect

```scala
val result: String = drunkFlip.unsafeRunSync()
val resultF: Future[String] = drunkFlip.unsafeToFuture()
```

# Cats Effect

The `IO[A]` hides the exact side effects that were performed. We can make them explicit using *Tagless Final* syntax

```
def drunkFlipF[F[_]: Random: [G[_]] =>> MonadError[G, String]]: F[String] =
  for {
    caught <- Random[F].nextBoolean
    heads <-
      if (caught) Random[F].nextBoolean
      else ApplicativeError[F, String].raiseError("We dropped the coin")
  } yield if (heads) "Heads" else "Tails"
```

The cognitive load is higher here 😱

# ZIO

- `ZIO[R, E, A]` introduces the error type `E` and dependencies `R` in the effect definition
  - It's still a monad on the `A` type ( `map` and `flatMap` )
  - It provides a *rich algebra* on the `ZIO` type to avoid monad transformers
  - It's a *referentially transparent* and *lazy* effect
  - It provides *structured concurrency* primitives
  - ...still a über effect

# ZIO

The `drunkFlip` function using `ZIO` effect is the following:

```
def drunkFlip: ZIO[Random, String, String] =
  for {
    caught <- Random.nextBoolean
    heads <-
      if (caught) Random.nextBoolean
      else ZIO.fail("We dropped the coin")
  } yield if (heads) "Heads" else "Tails"
```

**Effects are *explicit*** in the `R` type, and we can fail with *custom errors*

# ZIO

Running the effect means providing needed dependencies or *layers*

```
object Main extends ZIOAppDefault {
  override def run =
    drunkFlip.flatMap { result =>
      Console.printLine(result)
    }.provideLayer(ZLayer.succeed(RandomLive))
}
```

- We can use intersection type: `Random & Console`
- We must fulfill *all the dependencies* at once to run the effect

# Kyo: Meet Algebraic Effects

What if we can have types *listing Effect separately* and *handling* them virtually *once at a time*?

**Algebraic Effects and Handlers** do exactly that 🎉

- The type of the function tells us exactly what effects it uses
- **Kyo** is a novel library implementing Algebraic Effects

```
def drunkFlip: String < (IO & Abort[String]) = ???
```

# Kyo: Meet Algebraic Effects

- Each effect has its own *rich algebra* to describe the operations

```scala
import kyo.*

def drunkFlip: String < (IO & Abort[String]) = for {
  caught <- Random.nextBoolean
  heads  <- if (caught) Random.nextBoolean else Abort.fail("We dropped the coin")
} yield if (heads) "Heads" else "Tails"
```

- Kyo uses a *monad* to represent the effectful computation
  - We still have to use `flatMap` and `map`

# Kyo: Meet Algebraic Effects

We can decide to *handle each effect separately* (no über effect)

```scala
val partialResult: Result[String, String] < IO = Abort.run { drunkFlip }
```

- `Abort.run` is called an **Effect Handler**
  - It executes the `Abort` effect. The `IO` effect is *left untouched*
- Virtually, we can define our effect handler without changing the original recipe
  - For example, for testing purposes

# Build Your Own Effects System

# Build Your Own Effects System 🛠️

- All the effect systems we've seen are based on *monads* properties to *compose effectful functions*
  - They use *for-comprehension* style to give an imperative flavor to a sequence of `flatMap` and `map` calls

What if we could create an effect system that *doesn't rely on monads*, but almost preserves *referential transparency*?

😱 😱 😱 😱 😱 😱 😱 😱 😱 😱 😱 😱 😱

# Model the Effects' Algebra 🛠️

We'll focus on the `drunkFlip` example. We need effects that model

✔ non-determinism ( `Random` ),

✔ errors ( `Raise` )

```scala
trait Random {
  def nextBoolean: Boolean // <- Algebra of the effect
}
trait Raise[-E] { // <- `E` represents the error type
  def raise(error: => E): Nothing
}
```

# Access Std Library as an Effect

We need now to wrap the standard library with the effects

```scala
object Random {
  private val unsafe = new Random {
    override def nextBoolean: Boolean =
      scala.util.Random.nextBoolean()
  }
}
```

We call the variable `unsafe` ☣️ because it gives *direct, uncontrolled* access to the side effect

# Access Std Library as an Effect

We want to give tracked access to the side effects. Let's add some functions (a DSL) to our `object Random`

```
object Random {
  def nextBoolean(using r: Random): Boolean = r.nextBoolean
}
```

To generate a random `Boolean`, we need to *provide* an instance of the `Random` effect. We can call it a **capability**

- Calling `Random.nextBoolean` produces a *recipe* for the program

# Wrap It All Together

We have now all the bricks to build the `drunkFlip` function again 🙌

```
def drunkFlip(using Random, Raise[String]): String = {
    val caught = Random.nextBoolean
    val heads  =
      if (caught) Random.nextBoolean
      else Raise.raise("We dropped the coin")
    if (heads) "Heads" else "Tails"
  }
```

Is it magic 🪄? Variables `caught` and `heads` are treated as `Boolean`?! 🤯

# Welcome Context Functions 👋

- Scala 3 introduces **Context Functions**, fancy anonymous functions with only *implicit context parameters*

```scala
val program: (Raise[String], Random) ?=> String = drunkFlip
```

- Treated as **values** in contexts with the same implicit parameters
  - However, they are *recipes* to obtain the result

```scala
def drunkFlip(using Random, Raise[String]): String = {
  val caught: Boolean = Random.nextBoolean // 🤯
```

# Welcome Context Functions 👋

Behind the scenes, the Scala compiler rewrites the context function using a *surrogate type, not visible to the user*

```
trait ContextFunctionN[-T1, ..., -TN, +R]:
  def apply(using x1: T1, ..., xN: TN): R
```

Our `program` is rewritten as:

```
val program: new ContextFunction2[Raise[String], Random, String] {
  def apply(using Raise[String], Random): String = drunkFlip
}
```

# Handle the Effects

- Handlers are the structures that effectively *run* effectful functions

```scala
object Raise {
  def raise[E](error: => E)(using r: Raise[E]): Nothing = r.raise(error)
  def run[E, A](program: Raise[E] ?=> A): E | A =
    boundary {
      given unsafe: Raise[E] = new Raise[E] {
        override def raise(error: => E): Nothing = break(error)
      }
      program
    }
}
```

# Handle the Effects

- The Handler for the `Raise[E]` effect provides the `given` instance of the context parameter
  - We used the `boundary` and `break` functions to *control* the effect

```
val program: Random ?=> String | String = Raise.run { drunkFlip }
```

- The `Raise.run` handler *runs* the `Raise` effect, leaving the `Random` effect *untouched* 🥷
  - It's *curryfication*, but on a context parameters level

# Handle the Effects

- Changing the handler changes the *behavior* of the program
  - We can handle a `Raise[E] ?=> A` as an `Either[E, A]`

```scala
object Raise {
  def either[E, A](program: Raise[E] ?=> A): Either[E, A] =
    boundary {
      given r: Raise[E] = new Raise[E] {
        override def raise(error: => E): Nothing = break(Left(error))
      }
      Right(program)
    }
}
```

# Handle the Effects

Implementing the `Random` handler is relatively easy 👍

```
def run[A](program: Random ?=> A): A = program(using Random.unsafe)
```

We can even provide a test version of the `Random` effect

```
def test(fixed: Boolean)(program: Random ?=> Boolean) = {
  program(using new Random() {
    override def nextBoolean: Boolean = fixed
  })
}
```

# Handle the Effects

- We can run all the effects of the `drunkFlip` function *stacking* the handlers
  - We should do it at the *boundaries* of the system

```
val result: Either[String, String] = Random.run {
  Raise.either {
    drunkFlip
  }
}
```

...and we're done 🎉

# Properties of the Effect System

- We can say this Effect System uses a **Capability Passing Style**
- It implements the *Effect Pattern*
  - The type tells us the used *effects* and the type of the *result*
  - The execution is *deferred*

```scala
type Effect[E, A] = E ?=> A
```

- Handling effects at the *boundaries* of the system, we can use the **substitution model** again* 🚀

# Where's My `IO` Effect?

- Sometimes bad things happen. *Unpredictable* errors are thrown
- We want to execute an effectful function in a *dedicated process*

```scala
trait IO {}// Maybe Deferred would be a better name

object IO {
  def apply[A](program: => A): IO ?=> A = program
  def runBlocking[A](program: IO ?=> A): Try[A] = {
    val es: ExecutorService = Executors.newVirtualThreadPerTaskExecutor()
    Try { es.submit(() => program(using new IO {})).get() }
  }
}
```

# Where's My `IO` Effect?

- We can use Java Virtual Threads
  - Virtual Threads are implemented using *continuations*
  - They represent *fibers* 🧶, or *green threads* on the JVM
  - From Java 24, they are also safe for `synchronized` blocks 🎉
  - They support *structured concurrency* 🤝

```
val program: IO ?=> Int = IO {
  42  / 0
}
val result: Try[Int] = IO.runBlocking { program }
```

# Adding Monadic Operations

# Bonus Track

What if we can define `flatMap` and `map` in our Effect System 🤓?

We need to play some tricks. Let's define a class surrounding an effect and implement the `flatMap` and `map` functions on it

```
final class Effect[F](val unsafe: F)
object Effect {
  extension [F, A](eff: Effect[F] ?=> A) {
    inline def flatMap[B](inline f: A => Effect[F] ?=> B): Effect[F] ?=> B = f(eff)
    inline def map[B](inline f: A => B): Effect[F] ?=> B = eff.flatMap(a => f(a))
  }
}
```

# Bonus Track

We need to refactor the effects and the handlers accordingly (the refactor of the `Raise[E]` effect is omitted)

```
object Random {
  def nextBoolean(using r: Effect[Random]): Boolean = r.unsafe.nextBoolean

  def run[A](program: Effect[Random] ?=> A): A = program(using unsafe)

  val unsafe = new Effect(new Random {
    override def nextBoolean: Boolean = scala.util.Random.nextBoolean()
  })
}
```

# Bonus Track

We can rewrite the `drunkFlip` function using the new DSL:

```
def drunkFlip: (Effect[Random], Effect[Raise[String]]) ?=> String = for {
  caught <- Random.nextBoolean
  heads <-
    if (caught) Random.nextBoolean
    else Raise.raise("We dropped the coin")
} yield if (heads) "Heads" else "Tails"
```

If we substitute `inline` functions, we return to the version of `drunkFlip` that doesn't use the `Effect` class 🪄✨

# Conclusions and References

# Conclusions

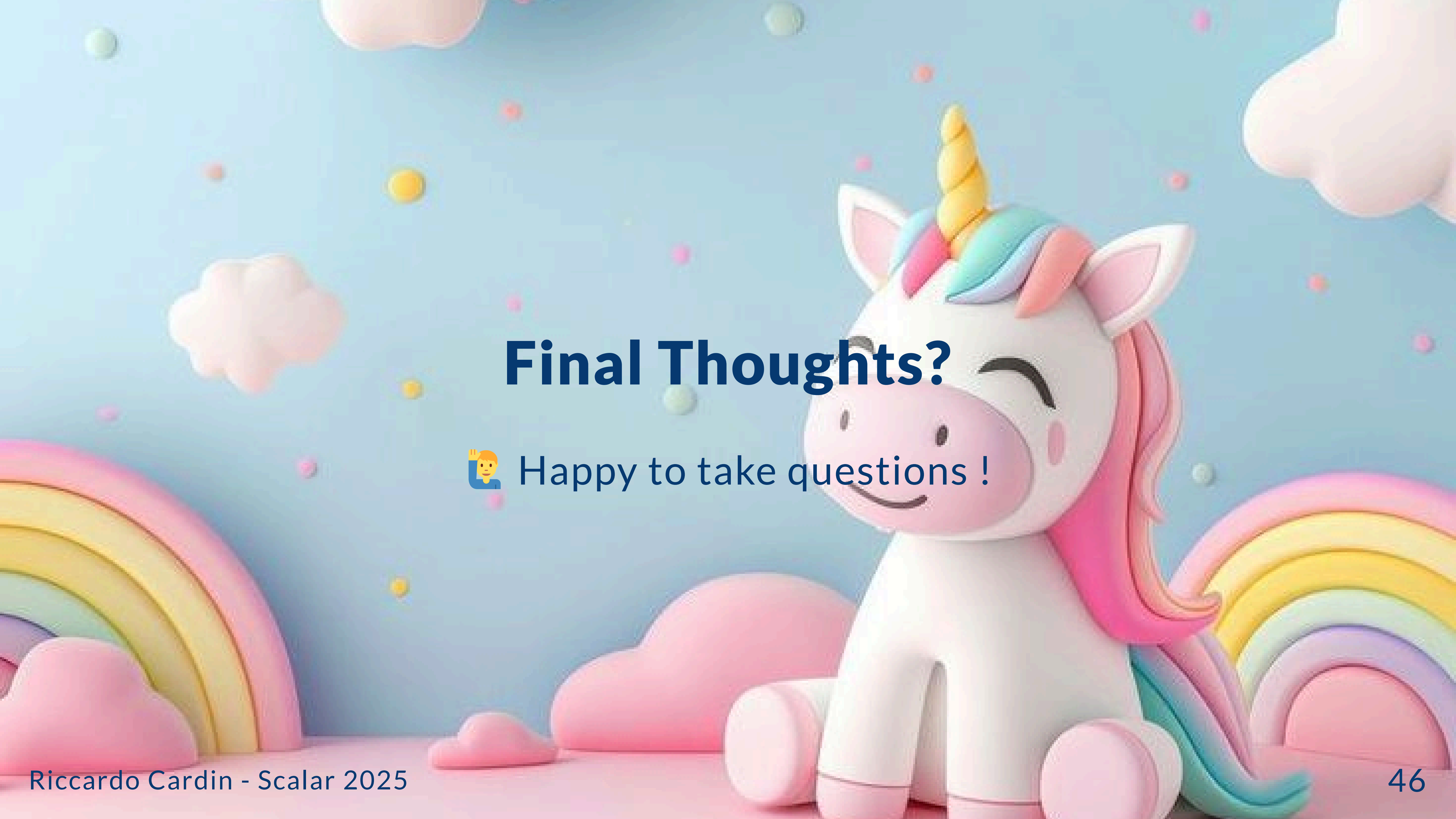
- We defined what is a *side effect* and why we don't like it
- We introduced the *Effect Pattern* and the *Effect Systems* to manage side effects in a controlled way
- We explored the *Cats Effect* and *ZIO* libraries as examples of *über effects*
- We introduced the *Kyo* library as an example of *Algebraic Effects*
- We built our own *Effect System* on top of *Context Functions*
- We saw how we can still define `flatMap` and `map` in our *Effect System*

# So Long, and

# Thanks for All the Fish 🐠!

👋

**YÆS**, *Yet Another Effect System*,
is a library implementing what we've seen today 😜



# Final Thoughts?

🙋‍♂️ Happy to take questions !

# References 📚


- Essential Effects, Adam Rosien
- Effect Oriented Programming, Bill Frasure, Bruce Eckel, James Ward
- Zionomicon, John A. De Goes, Adam Fraser, Milad Khajavi
- Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala, Jonathan Brachthäuser , Philipp Schuster, Klaus Ostermann

# References 📚


- Kyo, Streamlined Algebraic Effects, Simplified Functional Programming, Peak Scala Performance
- Scala 3 Context Functions
- The Ultimate Guide to Java Virtual Threads
- YÆS, Yet Another Effect System, An experimental effect system in Scala using capability passing style