

Introduction to Kyo

Adam Hearn

What is Kyo?

- Kyo is a powerful toolkit for developing with Scala
- Built from a series of standalone modules:
 - `kyo-data`: Low allocation, performant structures
 - `kyo-prelude`: Side-effect free **Algebraic effects**
 - `kyo-core`: Effects for IO, Async, & Concurrency
 - `kyo-scheduler`: high performance adaptive scheduler
 - `kyo-scheduler-zio`: boost your ZIO App!

What are Algebraic Effects?

What are Algebraic Effects?

- Effects
 - Descriptions of what you want
 - Produce what you want *when run*
 - Programs as values!
- Effects are backed by suspension
 - Suspension defers a computation until later
 - Separation of execution from definition:
 - Flexibility in execution (Retry, Delay, Interrupt)
 - Delayed implementation (Clock.live vs Clock.withTimeControl)

What are Algebraic Effects?

- Extensible & Composable Effects!
 - Fine-grained control over effect handling
 - Trivial combination of various abilities
 - Separation of effect declaration and implementation
 - User defined effects!
- Handlers: Define how effects are interpreted

Why use Algebraic Effects?

Why use Kyo?

- Includes flexible algebraic effects in **Scala**
- Designed for simplicity and performance
- Core effect handling is not restricted to included effects
 - User defined effects are

Kyo Syntax

```
val _: String < IO = IO("Hello scala.io!")
```

- Infix 'Pending' Type: Result < Effects
 - String < IO
- Effects are represented as unordered set:
 - File < (IO & Resource)'

Effects!!

IO: Side-Effect Suspension

```
object DB:
  def query[A](sql: SQL[A]): Chunk[A] < IO = ???

object Query:
  val _: Chunk[Person] < Any =
    import AllowUnsafe.embrace.danger
    IO.Unsafe.run(DB.query(sql"select * from person limit 5"))
```

- IO are handled individually (IO.Unsafe.run)
 - Unsafe APIs require an AllowUnsafe evidence

Abort: Short Circuit

```
case class User(email: String)
object User extends KyoApp:
  import UserError._
  enum UserError:
    case InvalidEmail
    case AlreadyExists

  def from(email: String): User < Abort[UserError] =
    if !email.contains('@') then Abort.fail(InvalidEmail)
    else User(email)

  val x: Unit < IO =
    Abort
      .run(from("adam@veak.co"))
      .map:
        case Result.Success(user)      => Console.println(s"Success! $user")
        case Result.Fail(InvalidEmail) => Console.println("Bad email!")
```

Env: Dependency Injection

```
abstract class Weather:
  def record(coordinates: Coordinates): Reading < IO

object Weather:
  val live: Weather < (Env[Drone] & Env[Sensor]) =
    for
      drone <- Env.get[Drone]
      sensor <- Env.get[Sensor]
    yield new Weather:
      def record(coordinates: Coordinates): Reading < IO =
        drone.fly(coordinates).andThen(sensor.read)
```

Kyo: Effect Widening

```
val a: String < IO = "Hello"  
val b: String < (IO & Abort[Exception]) = a  
val c: String < (IO & Abort[Exception] & Resource) = b
```

- Computations can be widened to include more effects
- Allows for flexible and composable code
- Plain values can be widened to Kyo computation
 - Widened values are not suspended
 - Widened values do not allocate ¹

¹Primitives widened to Kyo will box as Scala 3 does not support proper specialization

Kyo: Unnested encoding

```
object Write:
  def apply[S](v: String < S): Unit < (S & IO) =
    v.map(Buffer.write(_, "output.txt"))

object MyApp:
  val value: Unit < IO = Write("Hello, World!")
  val effect: Unit < (IO & Abort[IOException]) = Write(Console.readLine)
  val mapped: Unit < (IO & Abort[IOException]) = value.map(_ => effect)
```

- Widening pure values as effects enables fluent composition.
 - Functions can be defined to accept effects, and values can be passed in.
- `F.pure/ZIO.succeed` no more!

Kyo: Unnested encoding

- Effects can be easily combined using `map` and `flatMap`
- Resulting type includes all unique pending effects

Kyo: Effect Handling

```
val a: Int < Abort[Exception] = 42
val b: Result[Exception, Int] < Any = Abort.run(a)
val c: Result[Exception, Int] = b.eval
```

- Effects are handled explicitly
- Order of handling can affect the result type and value

KyoApp

Direct Syntax in Kyo

```
val a: String < (Abort[Exception] & IO) =  
  defer {  
    val b: String = await(IO("hello"))  
    val c: String = await(Abort.get(Right("world")))  
    b + " " + c  
  }
```

- defer and await provide a more intuitive syntax

Conclusion

- Kyo provides a powerful yet simple way to work with algebraic effects
- Offers composability, type safety, and performance
- Enables cleaner, more modular functional programming in Scala

Questions?

- !all effects suspend!
- Suspension + Handling
- ZIO each map/flatMap introduces suspension/resumption
- Kyo continues execution until an effect needs to be handled
<https://koka-lang.github.io/koka/doc/index.html>

Kyo supports a wide set of audience

- Library Authors
- Novices

Wrap up:

- Effects
- Kyo:
 - map vs flatMap
 - widening
 - IO/Abort/Async/Env/
 - Running your App

TODO: Kyo Data
kyo-scheduler