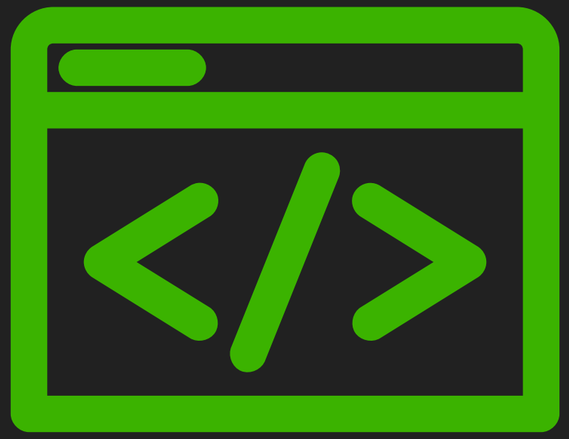
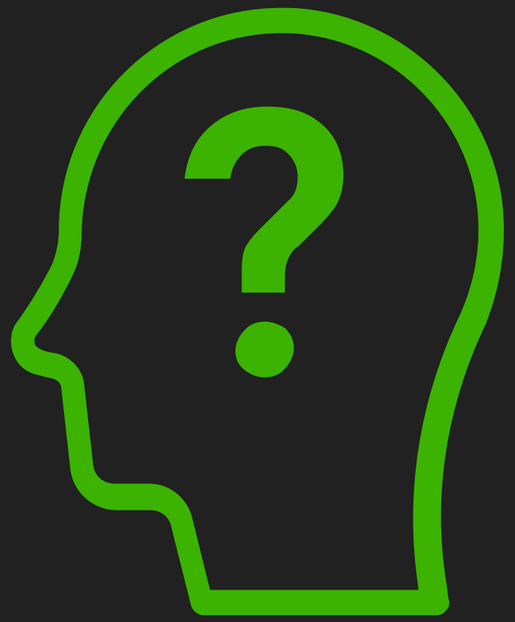
Macros and functional effects

Maxim Schuwalow
Functional Scala, 13.12.2019

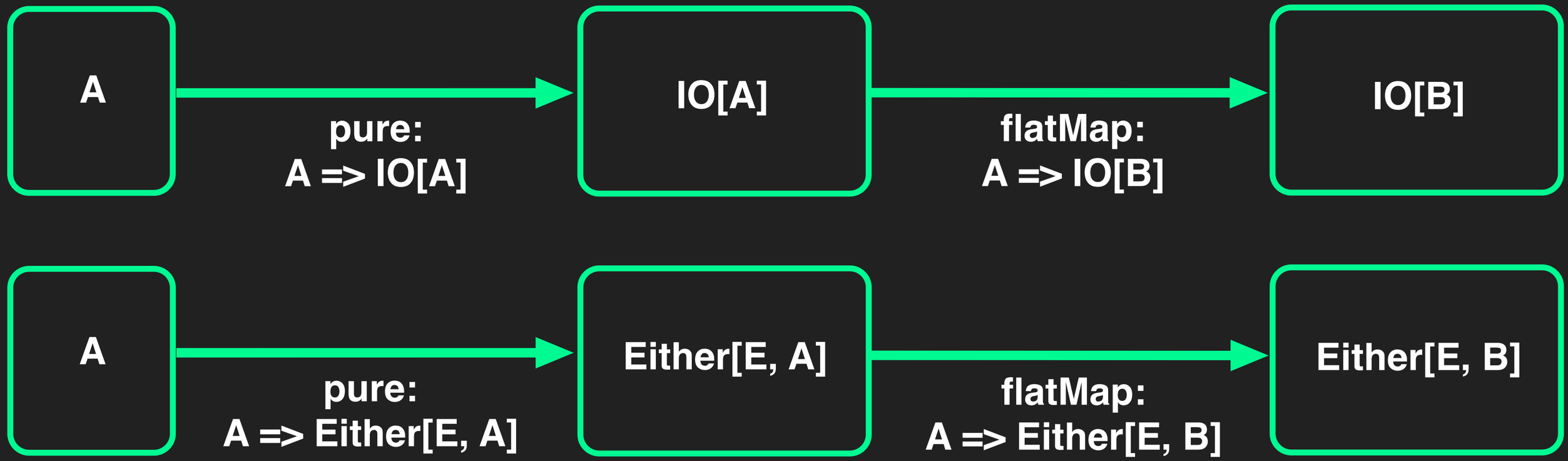
Agenda

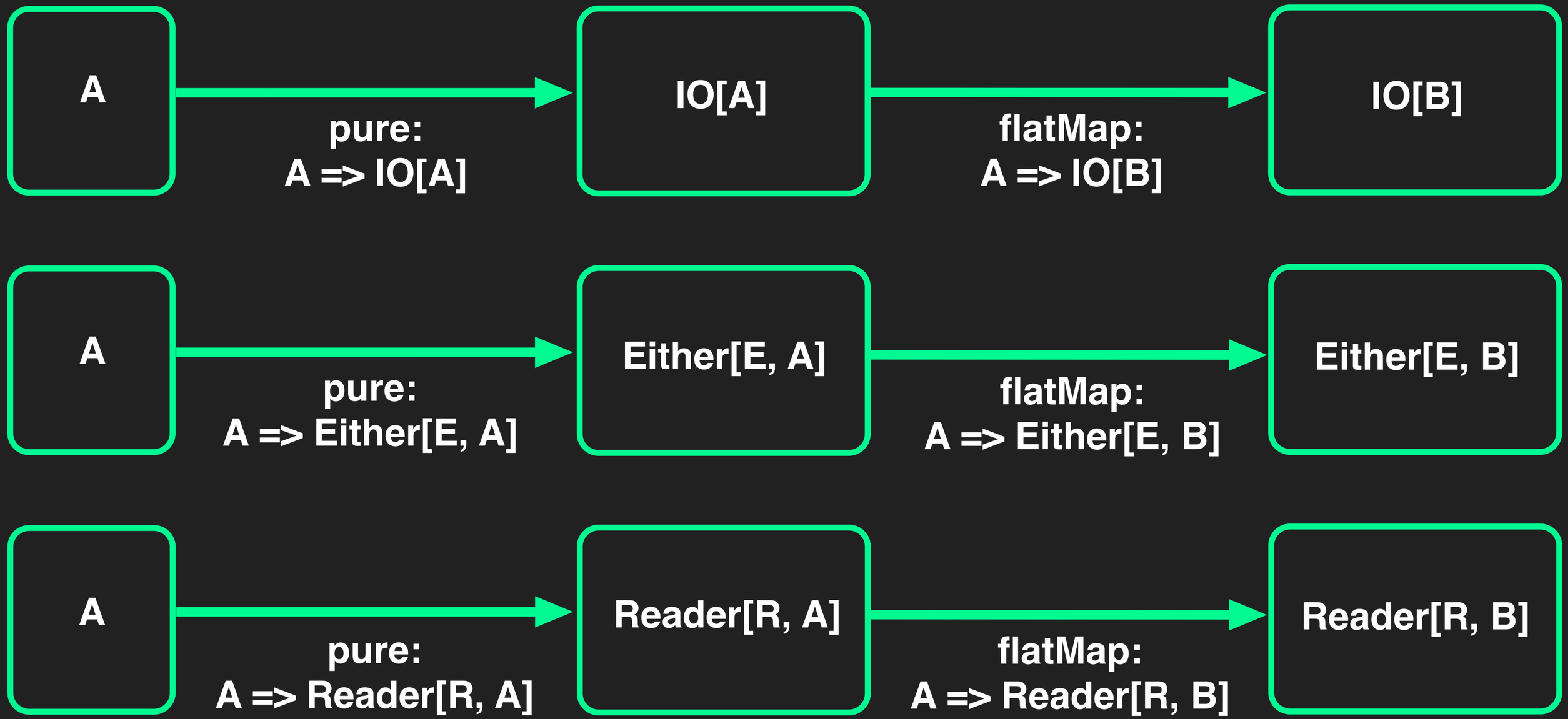
- Building applications with ZIO
- How can we do better?
- Introducing zio-macros






A
pure:
A => IO[A]
IO[A]
flatMap:
A => IO[B]
IO[B]


A
pure:
A => IO[A]
IO[A]
flatMap:
A => IO[B]
IO[B]
A
pure:
A => Either[E, A]
Either[E, A]
flatMap:
A => Either[E, B]
Either[E, B]


A
pure:
A => IO[A]
IO[A]
flatMap:
A => IO[B]
IO[B]
A
pure:
A => Either[E, A]
Either[E, A]
flatMap:
A => Either[E, B]
Either[E, B]
A
pure:
A => Reader[R, A]
Reader[R, A]
flatMap:
A => Reader[R, B]
Reader[R, B]

Module Pattern

```scala
trait Console extends Serializable {
  val console: Console.Service[Any]
}

object Console extends Serializable {
  trait Service[R] {
    def putStr(line: String): ZIO[R, Nothing, Unit]
    def putStrLn(line: String): ZIO[R, Nothing, Unit]
    val getStrLn: ZIO[R, IOException, String]
  }
}
```

```scala
import zio._
import zio.duration._
import zio.console._
import zio.clock._

val useBoth: ZIO[Clock with Console, Nothing, Unit] =
  for {
    _ <- ZIO.sleep(1.second)
    _ <- putStrLn("I waited like a good boy!")
  } yield ()

// bake the cake
val noDeps = ZIO[Any, Nothing, Unit]
  useBoth.provide(new Clock.Live with Console.Live {})
```

Okay,
let's build stuff...

Application #1

zio-keeper

```scala
trait Env[T] {
  val env: Env.Service[T]
}
object Env {
  trait Service[T] {
    val myself: T
    val activeView: TMap[T, ChunkConnection]
    val passiveView: TSet[T]
    val pickRandom: Int => STM[Nothing, Int]
    val cfg: Config
  }
}
```

```scala
trait Transport[T] {
  val transport: Transport.Service[Any, T]
}

object Transport {
  trait Service[R, T] {
    def send(to: T, data: Chunk[Byte]): ZIO[R, TransportError, Unit]
    def connect(to: T): ZIO[R, TransportError, ChunkConnection]
    def bind(addr: T): ZStream[R, TransportError, ChunkConnection]
  }
}
```

```
def handleJoin[T](
  msg: InitialProtocol.Join[T],
  con: ChunkConnection
): ZIO[Console with Env[T] with Transport[T], Error, Unit] =
  Env.using[T] { env =>
    for {
      others <- env.activeView.keys
                  .map(_.filterNot(_ == msg.sender)).commit
      _      <- ZIO.foreachPar_(others) { node =>
                  send(
                    node,
                    ForwardJoin(
                      env.myself, msg.sender,
                      TimeToLive(env.cfg.arwl)
                    )).ignore }
      _      <- addConnection(msg.sender, con)
    } yield ()
  }
```

```
object HyParView {
  def apply[R <: Transport[T] with Random with Console with Clock, T](
      localAddr: T,
      activeViewCapactiy: Int,
      passiveViewCapacity: Int,
      arwl: Int,
      prwl: Int,
      shuffleNActive: Int,
      shuffleNPassive: Int,
      shuffleTTL: Int,
      neighborSchedule: Schedule[R, Int, Any],
      shuffleSchedule: Schedule[R, Int, Any]
    ): ZManaged[R, Nothing, Membership[T]] = ???
}
```

```scala
type BaseEnv  = ???
type Env      = ???

def withFreshEnv[R <: BaseEnv, E, A](
  zio: ZIO[R with Env, E, A]
): ZIO[R, E, A] = ???
```

Application #2

Every doobie application, ever

```scala
type User = ???

trait UserRepository {
  val userRepository: UserRepository[Any]
}

object UserRepository {
  trait Service[R] {
    def addUser(request: CreationRequest): ZIO[R, Nothing, Unit]
    def deleteUser(user: User): ZIO[R, UserNotFound, Unit]
    def getUser(id: UserId): ZIO[R, Nothing, Option[User]]
  }
}
```

```scala
def doobieUserRepository(
  cfg: DBConfig
): ZManaged[Blocking, Throwable, UserRepository] =
  ZIO.runtime[Blocking].toManaged_.flatMap { implicit rt =>
    for {
      transactEC <- rt.Environment.blocking.blockingExecutor
                      .map(_.asEC).toManaged_
      connectEC   = rt.Platform.executor.asEC
      transactor <- HikariTransactor
                      .newHikariTransactor[Task](
                        cfg.driver, cfg.url, cfg.user,
                        cfg.password, connectEC,
                        Blocker.liftExecutionContext(transactEC)
                      ).toManaged
    } yield new DoobieUserRepository(transactor)
  }
```

```scala
val app: ZIO[UserRepository with Blocking with Console, Nothing, Unit] = ???
val dbConfig: DBConfig = ???

def main(args: List[String]) = {
  for {
    baseEnv                <- ZIO.environment[ZEnv]
    userRepository0        <- doobieUserRepository(dbConfig).provide(env)
    env = new UserRepository with Blocking with Console {
      val userRepository  = userRepository0.userRepository
      val blocking        = env.blocking
      val console         = env.console
    }
    _ <- app.provide(env)
  } yield ()
}
```

Application #3

```
final def delayedMEnv[R1 <: Clock](
  f: Duration => ZIO[R1, Nothing, Duration],
  g: (Clock.Service[Any] => Clock.Service[Any]) => R1 => R
): Schedule[R1, A, B] = ???
```

Requirements

1. Combinations of *R* are fully inferred ✔
2. Use larger *R* instead of smaller *R* ✔
3. (Effectfully) remove member to *R* ✘
4. (Effectfully) replace member in *R* ✘

Introducing zio-macros

sbt:

```
libraryDependencies += ("dev.zio" %% "zio-macros-core" % "0.6.0")
```

(A, B) => A with B

Mix[A, B]

```
trait Mix[A, B] {

  def mix(a: A, b: B): A with B
}

object Mix {

  def apply[A, B](implicit ev: Mix[A, B]) = ev

  implicit def deriveMix[A, B]: Mix[A, B] =
    macro Macros.mixImpl[A, B]
}
```

```scala
class Foo {
  def foo: Int = 2
}

trait Bar {
  def bar: Int
}

def withBar[A](a: A)(implicit ev: Mix[A, Bar]): A with Bar =
  ev.mix(a, new Bar { def bar = 2 })

withBar(new Foo()).bar // 2
```

EnrichWith

```
def enrichWith[A](a: A): EnrichWith[A] =
  new EnrichWith(a)

def enrichWithM[A]: EnrichWithM.PartiallyApplied[A] =
  new EnrichWithM.PartiallyApplied

def enrichWithManaged[A]: EnrichWithManaged.PartiallyApplied[A] =
  new EnrichWithManaged.PartiallyApplied
```

```scala
final class EnrichWithM[-R, +E, B](val zio: ZIO[R, E, B]) {
  ???

  def enrichZIO[R1, E1 >: E, A <: R](
    that: ZIO[R1, E1, A]
  )(implicit ev: A Mix B): ZIO[R1, E1, A with B] =
    that.flatMap(r1 => zio.provide(r1).map(ev.mix(r1, _)))
}

object EnrichWithM {
  final class PartiallyApplied[A] {
    def apply[R, E](zio: ZIO[R, E, A]): EnrichWithM[R, E, A] =
      new EnrichWithM(zio)
  }
}
```

```scala
val enrichClock =
  enrichWith[Clock](Clock.Live)

val enrichClockM =
  enrichWithM[Clock](ZIO.succeed(Clock.Live))

val enrichClockManaged =
  enrichWithManaged[Clock](ZManaged.succeed(Clock.Live))
```

```scala
val blockingWithDeps: ZIO[Console, Throwable, Blocking] =
  ZIO.succeed(Blocking.Live)

val enrichBlockingMWithDeps =
  enrichWithM[Blocking](blockingWithDeps)
```

```scala
implicit class ZIOOps[R, E, A](zio: ZIO[R, E, A]) {

  def @@[B](
    enrichWith: EnrichWith[B]
  )(implicit ev: A Mix B): ZIO[R, E, A with B] =
    enrichWith.enrichZIO[R, E, A](zio)

}
```

```scala
val enrichClock = enrichWith[Clock](Clock.Live)

val blockingWithDeps: ZIO[Console, Throwable, Blocking] =
  ZIO.succeed(Blocking.Live)

val enrichBlockingMWithDeps =
  enrichWithM[Blocking](blockingWithDeps)

//=============================================================

val randomWithClock: ZIO[Any, Nothing, Random with Clock] =
  ZIO.succeed(Random.Live) @@ enrichClock

val consoleWithBlocking: ZIO[Console, Nothing, Console with Blocking] =
  ZIO.environment[Console] @@ enrichBlockingMWithDeps
```

Coming soon:
Object-oriented optics

Patch

```
def patch[A, B](
  implicit ev: A Mix B
): (B => B) => A with B => A with B =
  f => old => ev.mix(old, f(old))
```

Patch

```
def patch[A, B](
  implicit ev: A Mix B
): (B => B) => A with B => A with B =
  f => old => ev.mix(old, f(old))

val mapClock: (Clock.Service[Any] => Clock.Service[Any]) => ZEnv => ZEnv =
  f => patch[ZEnv, Clock].apply(c => new Clock { val clock = f(c.clock) })

def disableLogging[R <: ZLogger: Mix[?, ZLogger], E, A](
  zio: ZIO[R, E, A]
): ZIO[R, E, A] =
  ZIO.provideSome[R](env => patch[R, ZLogger](_ => NoOpLogger)(env))
```

Local elimination

```scala
val zio: ZIO[Clock with Blocking, Nothing, Unit] = ???

val zioEliminated: ZIO[Blocking, Nothing, Unit] =
  zio.providePart[Clock](Clock.Live)
```

```scala
val zManaged: ZManaged[Clock with Blocking, Nothing, Unit] = ???

val zManagedEliminated: ZManaged[Blocking, Nothing, Unit] =
  zManaged.providePart[Clock](Clock.Live)
```

Polynote

polynote / polynote

Watch 77 | Unstar 3.3k | Fork 256

Code | Issues 114 | Pull requests 7 | Actions | Projects 0 | Wiki | Security | Insights

Branch: master **polynote** / polynote-env / src / main / scala / polynote / env / ops / **Enrich.scala** Find file | Copy path

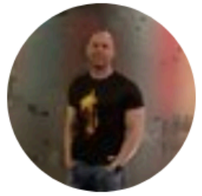

jeremyrsmith Basics working ee49d31 on Sep 5

1 contributor

17 lines (11 sloc) | 347 Bytes Raw | Blame | History

```
1  package polynote.env.ops
2
3  import polynote.env.macros.ZEnvMacros
4
5  trait Enrich[A, B] {
6    def apply(a: A, b: B): A with B
7  }
8
9  object Enrich {
10
11   implicit def enrichAny[B <: Any]: Enrich[Any, B] = new Enrich[Any, B] {
12     def apply(a: Any, b: B): Any with B = b
13   }
14
15   implicit def materialize[A, B]: Enrich[A, B] = macro ZEnvMacros.mkEnrich[A, B]
16
17 }
```

Is this magic?



John A De Goes
@jdegoes

Follow

Magic is a feature with non-compositional semantics that succeeds in making the common case easy, at the cost of making the uncommon cases surprising, impossible, or ridiculously complex. twitter.com/1ovthafew/stat …

Thank you!