# Kotlin + Λrrow for FP Scala Devs

# Lang Constructs

# Declarations

| Scala | Kotlin |
|---|---|
| val | val |
| var | var |
| object | object |
| trait | interface |
| class | class |
| def | fun |
| trait | interface |
| + | out |
| - | in |
| [A] | <A> |
| type | typealias |
| match | when |
| - | suspend (non blocking F<A> -> A) |
| - | with (scoped syntax) |

# Functions

Scala

```
object {
    def hello[A](a: A): String = s"hello $a"
}
```

Kotlin

```
fun <A> hello(a: A): String = "hello $a"
```

# Nullable types (Lang backed `Option`)

Scala

```
val maybeString: Option<String> = None
```

Kotlin

```
val maybeString: String? = null
```

# Data classes

Scala

```
case class Person(name: String, age: Int)
```

Kotlin

```
data class Person(val name: String, val age: Int)
```

# Sealed classes

## Scala

```
sealed abstract class ErrorType
case object MyError1 extends ErrorType
case object MyError2 extends ErrorType
case class MyError3(underlying: Throwable) extends ErrorType
```

## Kotlin

```
sealed class ErrorType
object MyError1 : ErrorType()
object MyError2 : ErrorType()
data class MyError3(val underlying: Throwable): ErrorType
```

# Pattern Matching

## Scala

```
errorType match {
    case MyError1 => ???
    case MyError2 => ???
    case MyError3(ex) => throw ex
}
```

## Kotlin

```
when (errorType) {
    is MyError1 => TODO()
    is MyError2 => TODO()
    is MyError3 => throw errorType.underlying //note smart cast
}
```

# object

Scala

```
object MySingleton
```

Kotlin

```
object MySingleton
```

# Companion object

Scala

```
case class Person(name: String, age: Int)
object Person {...}
```

Kotlin

```
data class Person(val name: String, val age: Int) {
    companion object { ... }
}
```

# Type aliases

Scala

```
type X = String
```

Kotlin

```
typealias X = String
```

# Path Dependent Types

Scala

```
abstract class Foo[A] {
  type X = A
}
```

Kotlin

```
abstract class Foo[A] {
  typealias X = A // will not compile
}
```

# Syntax Extensions

Scala

```
implicit class StringOps(value: String): AnyVal {
    def quote: String = s"--- $value"
}
```

Kotlin

```
fun String.quote(): String = "--- $value"
```

# Variance

Scala

```
class Option[+A] //A is covariant
class Functio1[-I, +O] // I is contravariant
class Leaf[A] // A is invariant
```

Kotlin

```
class Option<out A> //A is covariant
class Functio1n<in I, out O> // I is contravariant
class Leaf<A> // A is invariant
```

# Variance constrains

Scala

```
def run[A, B <: A]: Nothing = ???
def run[A, B >: A]: Nothing = ??? //No Kotlin equivalent
```

Kotlin

```
fun <A, B : A> run(): Nothing = TODO()
```

# FP Libraries

Scala : cats, scalaz
Kotlin: Λrrow

# Higher Kinded Types

Scala

```
class Service[F[_]]
```

Kotlin

```
class Service<F> // No way to express kind shape F<_>
```

# Higher Kinded Types Emulation

Scala

```
class Option[+A]
```

Kotlin

```
class ForOption private constructor()

typealias OptionOf<A> = arrow.Kind<ForOption, A>

inline fun <A> OptionOf<A>.fix(): Option<A> = this as Option<A>
class Option<out A> : Kind<OptionOf<A>>
```

# Higher Kinded Types Emulation

Kotlin

```
import arrow.higherkind

@higherkind class Option<out A> : OptionOf<A>
```

# Implicit Resolution

Scala

```
import cats.Functor

class Service[F[_]](implicit F: Functor[F]) {
    def doStuff: F[String] = F.pure("Hello Tagless World")
}

new Service[Option]
```

# Implicit Resolution (Runtime)

## Kotlin

```
import javax.inject
import arrow.*
import arrow.typeclasses.Functor

@typeclass interface Service<F> : TC {

    fun FF(): Functor<F>

    fun doStuff(): Kind<F, String> = FF().pure("Hello Tagless World")
}

val result: OptionOf<String> = service<ForOption>().doStuff()
van normalizedResult: Option<String> = result.fix()
```

# Implicit Resolution (Compile time)

## Kotlin

```
import javax.inject
import arrow.*
import dagger.*
import arrow.dagger.instances.*
import arrow.typeclasses.Functor

class Service<F> @Inject constructor(val FF: Functor<F>) {
    fun doStuff(): Kind<F, String> = FF.pure("Hello Tagless World")
}

@Singleton
@Component(modules = [ArrowInstances::class])
interface Runtime {
    fun optionService(): Service<ForOption>
    companion object {
        val implicits : Implicits = DaggerRuntime.create()
    }
}

val result: OptionOf<String> = Runtime.optionService().doStuff()
van normalizedResult: Option<String> = result.fix()
```

# Higher Kinded Types Partial application

Scala

```
Monad[Either[String, ?]]
```

Kotlin

```
Monad<EitherPartialOf<String>>
//Partial extensions are generated by @higherkind
```

# Cartesian Builder

## Scala

```
import cats.implicits._

case class Result(n: Int, s: String, c: Character)

(Option(1), Option("2"), Option('3')).mapN { case (n, s, c) =>
    Result(n, s, c)
}
```

# Cartesian Builder

## Kotlin

```
import arrow.*
import arrow.typeclasses.*

data class Result(val n: Int, val s: String, val c: Character)

Option.applicative().map(Option(1), Option("2"), Option('3'), {
    Result(n, s, c)
})
```

# Cartesian Builder

## Kotlin

```kotlin
import arrow.*
import arrow.typeclasses.*

@generic
data class Result(val n: Int, val s: String, val c: Character)

Option.applicative().mapToResult(Option(1), Option("2"), Option('3'))
// Option(Result(1, "2", '3'))
```

# Monad Comprehensions

Scala

```
for {
    a <- Option(1)
    b <- Option(1)
    x <- Option(1)
} yield a + b + c
//Option(3)
```

# Monad Comprehensions

## Kotlin

```
import arrow.*
import arrow.typeclasses.*

Option.monad().binding {
    val a = Option(1).bind()
    val b = Option(1).bind()
    val c = Option(1).bind()
    a + b + c
}
//Option(3)
```

# Arrow Monad Comprehensions

## Built atop Kotlin Coroutines

```
suspend fun <B> bind(m: () -> Kind<F, B>): B = suspendCoroutineOrReturn { c ->
    val labelHere = c.stackLabels // save the whole coroutine stack labels
    returnedMonad = flatMap(m(), { x: B ->
        c.stackLabels = labelHere
        c.resume(x)
        returnedMonad
    })
    COROUTINE_SUSPENDED
}
```

# Arrow Monad Comprehensions

```
binding

import arrow.*
import arrow.typeclasses.*

Try.monad().binding {
    val a = Try { 1 }.bind()
    val b = Try { 1 }.bind()
    a + b
}
//Success(2)
```

# Arrow Monad Comprehensions

```
binding

import arrow.*
import arrow.typeclasses.*

Try.monad().binding {
    val a = Try { 1 }.bind()
    val b = Try { 1 }.bind()
    a + b
}
//Success(2)
```

# Arrow Monad Comprehensions

## bindingCatch

```
import arrow.*
import arrow.typeclasses.*

Try.monad().bindingCatch {
    val a = Try { 1 }.bind()
    val b = Try<Int> { throw RuntimeException("BOOM") }.bind()
    a + b
}
//Failure(RuntimeException(BOOM))
```

# Arrow Monad Comprehensions

## bindingStackSafe

```
import arrow.*
import arrow.typeclasses.*

val tryMonad = Try.monad()

fun stackSafeTestProgram(n: Int, stopAt: Int): Free<ForTry, Int> =
    tryMonad.bindingStackSafe {
        val v = Try {n + 1}.bind()
        val r = if (v < stopAt) stackSafeTestProgram(M, v, stopAt).bind() else Try { v }.bind()
        r
    }

stackSafeTestProgram(0, 50000).run(tryMonad)
//Success(50000)
```

# Arrow Monad Comprehensions

## `binding(context: CoroutineContext)`

```
import arrow.*
import arrow.typeclasses.*

Try.monad().binding(UIThread) { //flatMap execution happens in the UI thread
    val a = IO { draw(1) }.bind()
    val b = IO { draw(1) }.bind()
    a + b
}
```

# Arrow already includes most things you need for FP

- Basic Data Types (Try, Eval, Option, NonEmptyList, Validated, ...)
- FP Type classes (Functor, Applicative, Monad...)
- Optics (Lenses, Prisms, Iso...)
- Generic (.tupled(), .tupleLabelled(), HListN, TupleN...)
- Effects (Async, Effect, IO, DeferredK, ObservableK...)
- MTL (FunctorFilter, TraverseFilter...)
- Free (Free, FreeApplicative...)

# Side projects

- Ank: Doc type checker like `tut`
- Helios: Json lib: Jawn port and based on arrow optics for its DSL
- Kollect: Fetch port
- Bow: Arrow for Swift

# Thanks for watching!

- @raulraja
- @47deg
- Λrrow
- Λrrow Tutorial Videos