Λrrow

Functional Data Types & Abstractions for Kotlin

About Me

Raul Raja

@raulraja

CTO @47deg

Started as...

Learning Exercise to learn FP over Slack

...then KΛTEGORY was born

Solution for Typed FP in Kotlin

Λrrow = KΛTEGORY + Funktionale

We merged with Funktionale to provide a single path to FP in Kotlin

Type classes

Arrow contains many FP related type classes

Error Handling	`ApplicativeError`, `MonadError`
Computation	`Functor`, `Applicative`, `Monad`, `Bimonad`, `Comonad`
Folding	`Foldable`, `Traverse`
Combining	`Semigroup`, `SemigroupK`, `Monoid`, `MonoidK`
Effects	`MonadDefer`, `Async`, `Effect`
Recursion	`Recursive`, `BiRecursive`,...
MTL	`FunctorFilter`, `MonadState`, `MonadReader`, `MonadWriter`, `MonadFilter`, ...

Data types

Λrrow contains many data types to cover general use cases.

Error Handling	`Option`, `Try`, `Validated`, `Either`, `Ior`
Collections	`ListKW`, `SequenceKW`, `MapKW`, `SetKW`
RWS	`Reader`, `Writer`, `State`
Transformers	`ReaderT`, `WriterT`, `OptionT`, `StateT`, `EitherT`
Evaluation	`Eval`, `Trampoline`, `Free`, `FunctionN`
Effects	`IO`, `Free`, `ObservableKW`
Optics	`Lens`, `Prism`, `Iso`,...
Recursion	`Fix`, `Mu`, `Nu`,...
Others	`Coproduct`, `Coreader`, `Const`, ...

A few syntax examples

```
import arrow.*
import arrow.core.*
import arrow.intro.* // slides stuff

Option(1).map { it + 1 }
// Some(2)
```

A few syntax examples

```
Try<Int> { throw RuntimeException("BOOM!") }.map { it + 1 }
// Failure(exception=java.lang.RuntimeException: BOOM!)
```

A few syntax examples

```
val x = Right(1)
val y = 1.right()
x == y
// true
```

Applicative Builder

```
import arrow.instances.*

data class Profile(val id: Long, val name: String, val phone: Int)

fun profile(maybeId: Option<Long>, maybeName: Option<String>, maybePhone: Option<Int>): Option<Profile> =
  ForOption extensions {
    map(maybeId, maybeName, maybePhone, { (a, b, c) ->
       Profile(a, b, c)
    }).fix()
  }

profile(1L.some(), "William Alvin Howard".some(), 555555555.some())
// Some(Profile(id=1, name=William Alvin Howard, phone=555555555))
```

Applicative Builder

```
@generic
data class Profile(val id: Long, val name: String, val phone: Int) {
  companion object
}

fun profile(maybeId: Option<Long>, maybeName: Option<String>, maybePhone: Option<Int>): Option<Profile> =
  mapToProfile(maybeId, maybeName, maybePhone).fix()

profile(1L.some(), "William Alvin Howard".some(), 555555555.some())
// Some(Profile(id=1, name=William Alvin Howard, phone=555555555))
```

Applicative Builder (Same for all data types)

```
@generic
data class Profile(val id: Long, val name: String, val phone: Int) {
  companion object
}

fun profile(tryId: Try<Long>, tryName: Try<String>, tryPhone: Try<Int>): Try<Profile> =
  mapToProfile(tryId, tryName, tryPhone).fix()

profile(Try { 1L }, Try { "William Alvin Howard" }, Try { 555555555 })
// Success(Profile(id=1, name=William Alvin Howard, phone=555555555))
```

Comprehensions - Vanilla

Generalized to all monads. A suspended function provides a non blocking `F<A> -> A`

```
import arrow.typeclasses.*

fun profile(maybeId: Option<Long>,
            maybeName: Option<String>,
            maybePhone: Option<Int>): Option<Profile> =
  ForOption extensions {
   binding { // <-- `coroutine starts`
     val id = maybeId.bind() // <-- `suspended`
     val name = maybeName.bind() // <-- `suspended`
     val phone = maybePhone.bind() // <-- `suspended`
     Profile(id, name, phone)
   }.fix() // <-- `coroutine ends`
  }

profile(2L.some(), "Haskell Brooks Curry".some(), 555555555.some())
// Some(Profile(id=2, name=Haskell Brooks Curry, phone=555555555))
```

Comprehensions - Exception Aware

Automatically captures exceptions for instances of `MonadError<F, Throwable>`

```
ForTry extensions {
  bindingCatch {
      val a = Try { 1 }.bind()
      val b = Try { 1 }.bind()
      throw RuntimeException("BOOM") // <-- `raises errors to MonadError<F, Throwable>`
      val c = Try { 1 }.bind()
      a + b + c
  }
}
// Failure(exception=java.lang.RuntimeException: BOOM)
```

Comprehensions - Filterable

Imperative filtering control for data types that can provide `empty` values.

```
import arrow.data.*
import arrow.mtl.typeclasses.*
import arrow.mtl.instances.*

fun <F> MonadFilter<F>.continueIfEven(fa: Kind<F, Int>): Kind<F, Int> =
  bindingFilter {
    val v = fa.bind()
    continueIf(v % 2 == 0)
    v + 1
  }
```

```
ForOption extensions { continueIfEven(Option(2)) }
// Some(3)
```

```
ForListK extensions { continueIfEven(listOf(2, 4, 6).k()) }
// ListK(list=[3, 5, 7])
```

Integrations - Rx2

Let’s take an example and convert it to a comprehension.

```
getSongUrlAsync()
  .flatMap { MediaPlayer.load(it) }
  .flatMap {
    val totalTime = musicPlayer.getTotaltime()
    audioTimeline.click()
      .map { (timelineClick / totalTime * 100).toInt() }
   }
```

Integrations - Rx2

Arrow provides `MonadError<F, Throwable>` for `Observable`

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

ForObservableK extensions {
 bindingCatch {
   val songUrl = getSongUrlAsync().bind()
   val musicPlayer = MediaPlayer.load(songUrl)
   val totalTime = musicPlayer.getTotaltime()
   (audioTimeline.click().bind() / totalTime * 100).toInt()
 }
}
```

Integrations - Kotlin Coroutines

Arrow provides `MonadError<F, Throwable>` for `Deferred`

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

ForDeferredK extensions {
 bindingCatch {
   val songUrl = getSongUrlAsync().bind()
   val musicPlayer = MediaPlayer.load(songUrl)
   val totalTime = musicPlayer.getTotaltime()
   (audioTimeline.click().bind() / totalTime * 100).toInt()
 }
}
```

Transforming immutable data

Arrow includes an `optics` library that make working with immutable data a breeze

```
data class Street(val number: Int, val name: String)
data class Address(val city: String, val street: Street)
data class Company(val name: String, val address: Address)
data class Employee(val name: String, val company: Company)
```

Transforming immutable data

Λrrow includes an `optics` library that make working with immutable data a breeze

```
val employee = Employee("John Doe",
                Company("Λrrow",
                 Address("Functional city",
                   Street(23, "lambda street"))))
employee
// Employee(name=John Doe, company=Company(name=Λrrow, address=Address(city=Functional city, street=Street(number=23,
```

Transforming immutable data

while `kotlin` provides a synthetic `copy` dealing with nested data can be tedious

```
employee.copy(
  company = employee.company.copy(
    address = employee.company.address.copy(
      street = employee.company.address.street.copy(
        name = employee.company.address.street.name.capitalize()
      )
    )
  )
)
// Employee(name=John Doe, company=Company(name=Λrrow, address=Address(city=Functional city, street=Street(number=23,
```

Optics without boilerplate

You may define composable `Lenses` to work with immutable data transformations

```
val employeeCompany: Lens<Employee, Company> = Lens(
        get = { it.company },
        set = { company -> { employee -> employee.copy(company = company) } }
)

val companyAddress: Lens<Company, Address> = Lens(
        get = { it.address },
        set = { address -> { company -> company.copy(address = address) } }
)
...
```

Optics without boilerplate

You may define composable `Lenses` to work with immutable data transformations

```
import arrow.optics.*

val employeeStreetName: Lens<Employee, String> =
  Employee.company compose Company.address compose Address.street compose Street.name

employeeStreetName.modify(employee, String::capitalize)
// Employee(name=John Doe, company=Company(name=Arrow, address=Address(city=Functional city, street=Street(number=23,
```

Optics without boilerplate

Or just let Arrow `@optics` do the dirty work

```
+ @optics data class Employee(val name: String, val company: Company)
- val employeeCompany: Lens<Employee, Company> = Lens(
-        get = { it.company },
-        set = { company -> { employee -> employee.copy(company = company) } }
- )
-
- val companyAddress: Lens<Company, Address> = Lens(
-        get = { it.address },
-        set = { address -> { company -> company.copy(address = address) } }
- )
- ...
```

Optics without boilerplate

Optics comes with a succinct and powerful DSL to manipulate deeply nested immutable properties

```
import arrow.optics.dsl.*

Employee.company.address.street.name.modify(employee, String::toUpperCase)
// Employee(name=John Doe, company=Company(name=Λrrow, address=Address(city=Functional city, street=Street(number=23,
```

Optics without boilerplate

You can also define `@optics` for your sealed hierarchies

```
@optics sealed class NetworkResult
@optics data class Success(val content: String): NetworkResult()
@optics sealed class NetworkError : NetworkResult()
@optics data class HttpError(val message: String): NetworkError()
object TimeoutError: NetworkError()
```

Optics without boilerplate

Where you operate over sealed hierarchies manually...

```
val networkResult: NetworkResult = HttpError("boom!")
val f: (String) -> String = String::toUpperCase

val result = when (networkResult) {
  is HttpError -> networkResult.copy(f(networkResult.message))
  else -> networkResult
}

result
// HttpError(message=BOOM!)
```

Optics without boilerplate

...you cruise now through properties with the new optics DSL

```
NetworkResult.networkError.httpError.message.modify(networkResult, f)
// HttpError(message=BOOM!)
```

In the works

arrow-generic	Generic programming with products, coproducts and derivation
arrow-streams	A functional `Stream<F, A>` impl that abstract over F and complementes `arrow-effect`
arrow-android	FP solutions to common Android issues `Ex: Activity lifecycle`

Λrrow is modular

Pick and choose what you'd like to use.

Module	Contents
typeclasses	`Semigroup`, `Monoid`, `Functor`, `Applicative`, `Monad`…
core/data	`Option`, `Try`, `Either`, `Validated`…
effects	`IO`
effects-rx2	`ObservableKW`, `FlowableKW`, `MaybeK`, `SingleK`
effects-coroutines	`DeferredK`
mtl	`MonadReader`, `MonadState`, `MonadFilter`,…
free	`Free`, `FreeApplicative`, `Trampoline`,…
recursion-schemes	`Fix`, `Mu`, `Nu`
optics	`Prism`, `Iso`, `Lens`,…
meta	`@higherkind`, `@deriving`, `@instance`, `@optics`

Kotlin limitations for Typed FP

Kotlin limitations for Typed FP

Emulated Higher Kinds through Lightweight higher-kinded Polymorphism

Kotlin limitations for Typed FP

Fear not, `@higherkind`'s got your back!

```
+ @higherkind sealed class Option<A> : OptionOf<A>
- class ForOption private constructor() { companion object }
- typealias OptionOf<A> = Kind<ForOption, A>
- inline fun <A, B> OptionOf<A>.fix(): Option<A> = this as Option<A>
```

Λrrow ad-hoc polymorphism

With emulated Higher Kinds and Type classes we can now write polymorphic code

```
import arrow.Kind
import arrow.core.*
import arrow.effects.*
import arrow.typeclasses.*
```

Λrrow ad-hoc polymorphism

With emulated Higher Kinds and Type classes we can now write polymorphic code

```
interface Service1<F> : Functor<F> {
  fun Kind<F, Int>.addOne(): Kind<F, Int> =
    map { it + 1 }
}
```

Λrrow ad-hoc polymorphism

With emulated Higher Kinds and Type classes we can now write polymorphic code

```
interface Service2<F> : Functor<F> {
  fun Kind<F, Int>.addTwo(): Kind<F, Int> =
    map { it + 2 }
}
```

Λrrow ad-hoc polymorphism

With emulated Higher Kinds and Type classes we can now write polymorphic code

```
interface App<F> : Service1<F>, Service2<F> {
  fun Kind<F, Int>.addThree(): Kind<F, Int> =
    addOne().addTwo()
}
```

Λrrow ad-hoc polymorphism

With emulated Higher Kinds and Type classes we can now write polymorphic code

```
/* Our app works for all functors */
fun <F, A> Functor<F>.app(f: App<F>.() -> A): A =
  f(object : App<F> {
    override fun <A, B> Kind<F, A>.map(f: (A) -> B): Kind<F, B> = this@app.run { map(f) }
  })
```

Λrrow ad-hoc polymorphism

Program that are abstract and work in many runtimes!

```
ForOption extensions {
  app {
    Option(1).addThree()
  }
}
// Some(4)
```

Arrow ad-hoc polymorphism

Program that are abstract and work in many runtimes!

```
ForTry extensions {
  app {
    Try { 1 }.addThree()
  }
}
// Success(value=4)
```

Arrow ad-hoc polymorphism

Program that are abstract and work in many runtimes!

```
ForIO extensions {
  app {
    IO { 1 }.addThree().fix().unsafeRunSync()
  }
}
// 4
```

Type Classes

This is how you define Type Classes in Arrow (for now)

```
interface Functor<F> : Typeclass {
  fun <A, B> map(fa: HK<F, A>, f: (A) -> B): HK<F, B>
}
```

Implementing type class instances is easy...

@deriving

Λrrow can derive instances based on conventions in your data types

```
@higherkind
@deriving(
        Functor::class,
        Applicative::class,
        Monad::class,
        Foldable::class,
        Traverse::class,
        TraverseFilter::class,
        MonadFilter::class)
sealed class Option<out A> : OptionKind<A> {
   ...
}
```

@instance

Λrrow allows you to hand craft instances

```
@instance(Either::class)
interface EitherFunctorInstance<L> : Functor<EitherKindPartial<L>> {
    override fun <A, B> map(fa: EitherKind<L, A>, f: (A) -> B): Either<L, B> =
    fa.ev().map(f)
}
//Either.functor<L>() is available after @instance is processed
```

KEEP-87

But we are not stopping here, we want to get rid of some of the codegen.

KEEP-87 is A KEEP to introduce Type Classes in Kotlin!

https://github.com/Kotlin/KEEP/pull/87

KEEP-87

Type Classes & Instances

```
extension interface Monoid<A> {
  fun A.combine(b: A): A
  val empty: A
}

extension object IntMonoid : Monoid<Int> {
  fun Int.combine(b: Int): Int = this + b
  val empty: Int = 0
}
```

KEEP-87

Declaration site

```
fun combineOneAndTwo(with Monoid<Int>) =
  1.combine(2) // `this` is an instance of `Monoid<Int>`
```

Desugars to

```
fun combineOneAndTwo(ev: Monoid<Int>) =
  with(ev) { 1.combine(2) } // `this` is ev
```

KEEP-87

Call site

```
import IntMonoid
combineOneAndTwo() // instance is resolved via imports and injected by the compiler
```

Desugars to

```
import IntMonoid
combineOneAndTwo(IntMonoid) // compatible with java and allows explicit overrides
```

An ecosystem of libraries

Ank

Markdown documentation, verification and snippet evaluator for Kotlin

An ecosystem of libraries

Helios

A fast, purely functional JSON lib for Kotlin

An ecosystem of libraries

Kollect

Efficient data access with id dedup, parallelization, batching and caching.

Credits

Λrrow is inspired in great libraries that have proven useful to the FP community:

- Cats
- Scalaz
- Freestyle
- Monocle
- Funktionale
- Paguro

72 Contributors and counting

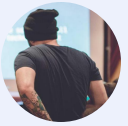

anstaendig


arturogutierrez


ffgiraldez


Guardiola31337

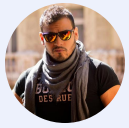

javipacheco


JMPergar


JorgeCastilloPrz


jrgonzalezg


nomisRev


npatarino


pablisco

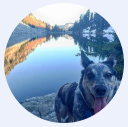

pakoito


pedrovgs


pt2121

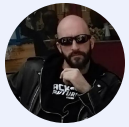

raulraja


wiyarmir

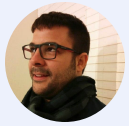

andyscott


Atternatt


calvellido


dominv

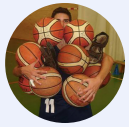

GlenKPeterson


israelperezglez


sanogueralorenzo


Takhion


victorg1991


tonilopezmr


NigelHeylen


ersin-ertan

Join us!

Github	https://github.com/arrow-kt/arrow
Slack	https://kotlinlang.slack.com/messages/C5UPMM0A0
Gitter	https://gitter.im/kategory/Lobby

We provide 1:1 mentoring for both users & new contributors!

Thanks!

Thanks to everyone that makes Λrrow possible