Introduction to Kategory

1. Kategory

A companion library to the stdlib enabling Typed FP in Kotlin with functional datatypes, typeclasses and more.

- DataTypes: (Option, Try, Either, EitherT, Free,...)
- Typeclasses : (Functor, applicative, Monad, ...)
- Utils: (comprehensiones monadicas, applicative builder,...)

Index

- What is Kategory?
- What is Typeclass?
- Common FP typeclasses
- Option, a datatype, instances and utilities in detail
- Kategory programming styles
- Other useful datatypes and abstractions
- Contribute to Kategory

2. What is a Typeclass?

Like an interface but parametric to a data type.

```
interface Semigroup<A> {
  fun append(a: A, b: A): A
}

object StringSemigroup: Semigroup<String> {
  fun append(a: String, b: String): String = a + b
}

object IntSemigroup: Semigroup<Int> {
  fun append(a: Int, b: Int): Int = a + b
}

fun <A> append(a: A, b: A, SM : Semigroup<A>): A = SM.append(a, b)

append("a", "b", StringSemigroup) // "ab"
append(1, 1, IntSemigroup) // 2
```

3. Common FP Typeclasses

Semigroup

Used to reduce or append values of the same type

```
interface Semigroup<A> {
  fun append(a: A, b: A): A
}
```

3. Common FP Typeclasses

Monoid (Semigroup + empty value)

```
interface Monoid<A> : Semigroup<A> {
  fun empty(): A
}
```

3. Common FP Typeclasses

Functor

Transform the value from A to B that's inside a type constructor

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

Functor<Option.F>.map(Option(1), { it + 1 }) // Option(2)

inline fun <reified F, A, B>
       transform(fa: HK<F, A>, f: (A) -> B, FT: Functor<F> = functor()) : HK<F, B> =
       FT.map(fa, f)

transform(Option(1), { it + 1}) //Option(2)
transform(List(1), {it + 1}) // List(2)
transform(Future(1), {it + 1}) // Future(2)
```

3. Common FP Typeclasses

Applicative

Computations that are independent from each other

```
interface Applicative<F> : Functor<F> {
  fun <A> pure(a: A): HK<F, A>
  fun <A, B> ap(fa: HK<F, A>, ff: HK<F, (A) -> B>): HK<F, B>
  fun <A, B> product(fa: HK<F, A>, fb: HK<F, B>): HK<F, Tuple2<A, B>>
  ...
}

inline fun <reified F, A, B>
       join(fa: HK<F, A>, fb: HK<F, B>, AP: Applicative<F> = applicative()) : HK<F, Tuple2<A, B>> =
       FT.product(fa, fb)

join(Option("a"), Option(1)) // Option("a", 1)
join(Future("a"), Future(1)) // Future("a", 1)
```

3. Common FP Typeclasses

Monad

Dependent and sequential computations

```
interface Monad<F> : Applicative<F> {
  fun <A, B> flatMap(fa: HK<F, A>, f: (A) -> HK<F, B>): HK<F, A>
}

inline fun <reified F> compute(M: Monad<F> = monad()) : HK<F, Int> =
   M.binding {
     val x = M.pure(1)
     val y = M.pure(x + 1)
     yields(x + y)
   }

compute(Option) // Option(2)
compute(List) // List(2)
compute(Future) // Future(2)
```

3. Common FP Typeclasses

Traverse

Iterate over structures applying functions

```
interface Traverse<F> : Foldable<F> {
  fun <G, A, B> traverse(fa: HK<F, A>, f: (A) -> HK<G, B>, GA: Applicative<G>): HK<G, HK<F, B>>
}
```

```
List(Option(1), Option(2), Option(3)).sequence() // Option(List(1,2,3))
List(Future(1), Future(2), Future(3)).sequence() // Future(List(1,2,3))
```

3. Common FP Typeclasses

(Reduccion) : Semigroup -> Monoid

(Transformacion) : Functor -> Applicative -> Monad

(Iteracion) : Foldable -> Traversable

4. Source zoom on Option

A complete source code walthrough over Option instances and usages examples.

- Main combinators
- Instances
- Monadic computation
- Applicative computation
- Kategory code org and conventions

5. Kategory programming styles

FPers usually follow a combination of several styles including and not limited to:

- Concrete programming with datatypes through syntax
- Monad Transformers
- Tagless (Typeclasses)
- Free / Interpreters

5. Kategory programming styles

Concrete programming with datatypes + transformers where needed

```
typealias Result<A> = Future<Either<Throwable, A>>

fun <A> attempt(op: () -> A): Result<A> = Future {
    Try(op).fold({ e ->
        Either.Left(e)
    }, { ok ->
        Either.Right(ok)
    })
}

attempt("x".toInt()) // Future(Left(IllegalArgumentException))
```

5. Kategory programming styles

Tagless or abstract style with typeclasses

```
typealias Result<A> = Future<Either<Throwable, A>>

inline fun <reified F, A> attempt(op: () -> A, ME: MonadError<Throwable, A> = monadError()): HK<F, A> =
  ME.catch(op)

attempt<Result.F, Int>("x".toInt()) // Future(Left(IllegalArgumentException))
attempt<Either.F, Int>("x".toInt()) // Left(IllegalArgumentException)
attempt<Try.F, Int>("x".toInt()) // Failure(IllegalArgumentException)
```

5. Kategory programming styles

Free / Interpreters

Deserves it's own session. It allows to decouple program declaration from interpretation.

6. Many more useful datatypes / abstractions

- **Either** : Error handling and multi result values
- **Validated** : Data validation with error accumulation
- **Try** : Capturing Exceptions from unknown computations
- **Reader** : Defer evaluation with arguments until the end. Dependency injection
- **Writer** : Accumulate values (log)

7. Contribute to kategory

Issue tracker + code overview

Kategory brought to you by...

@pakoito

@raulraja

@JMPergar

@JorgeCastilloPrz

@ffgiraldez

@jrgonzalezg

@aballano

@arturogutierrez

@sanogueralorenzo

@wiyarmir

Thanks!