
47
47 DEGREES
λcademy
TALK
FUNCTIONAL ANDROID DEVELOPMENT
Jorge_Castillo
@JorgeCastilloPr

Why FP? 🤔

Achieve **determinism** to fight race conditions.

- Pure functions 👉 improve code reasoning.
- Keep side effects under control.

Concern separation

Staying **declarative and deferred**.

- Memory representation of the program (algebras).
- Decoupled runtime - optimizations.
- Simple example: Kotlin `Sequences` 👉 terminal ops to consume - `toList()`

Another example of this?

Jetpack Compose 😲

Compose

Also applies **concern separation**

- Creates an in-memory representation of the UI tree 🌲
- The runtime interprets it by applying desired optimizations.

  (Run composable functions in parallel, in different order, smart recomposition...).

Composable functions

Similar to suspend functions 🤔

- Description of an effect to render UI.
- Only callable from within other composable functions or a prepared **environment** 👉 integration point 👉 `setContent {}`
- Enforces a usage scope to keep control over it.

Composable functions

```
class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    /* ... */
    setContent { // integration point for Android
      AppTheme {
        MainContent() // composable tree
      }
    }
  }
}
```

- The integration point interprets the in-memory UI tree 👉 skia in Android
- Allow using different runtimes.

Android architecture?

We can leverage the same idea using **suspend** 🍃.

- Flags a potentially blocking long running computation 👉 effect 🌀
- Enforces it to run under a prepared environment (Coroutine).
- Makes the effect compile **time tracked**.

Flag effects as suspend

Make 'em **pure**!

```
interface UserService {
-   fun loadUser(): User
+   suspend fun loadUser(): User
}
```

```
class UserPersistence {
-   fun loadUser(): User = TODO()
+   suspend fun loadUser(): User = TODO()
}
```

```
class AnalyticsTracker {
-   fun trackEvent(event: Event): Unit = TODO()
+   suspend fun trackEvent(event: Event): Unit = TODO()
}
```

But we'll need a runtime

Every suspended program **requires an environment** (runtime) to run.

- Suspend makes our program declarative 👉 description of effects.
- Crawls the call stack up until the integration point 👉 coroutine launch.

Environment in KotlinX

- KotlinX Coroutines builders 👉 launch, async.

```
class MyFragment: Fragment() {
  override fun onViewCreated(...) {
    /* ... */
    viewLifecycleOwner.lifecycleScope.launch {
      // suspended program 🍃
    }
  }
}
```

Environment in FP

- Arrow Fx Coroutines Environment.

```
class MyFragment: Fragment() {
  override fun onViewCreated(...) {
    /* ... */
    val env = Environment()
    val cancellable = env.unsafeRunAsyncCancellable(
     { /* suspended program */ },
     { e -> /* handle errors unhandled by the program */ },
     { a -> /* handle result of the program */ }
    )
  }
}
```

- Takes care of the execution strategy / context to run the program.

App entry points

Also called **"edge of the world"**.

- Android 🙅 no `suspend` entry points.
- Inversion of control.
- Lifecycle callbacks 👉 entry points to hook logic.

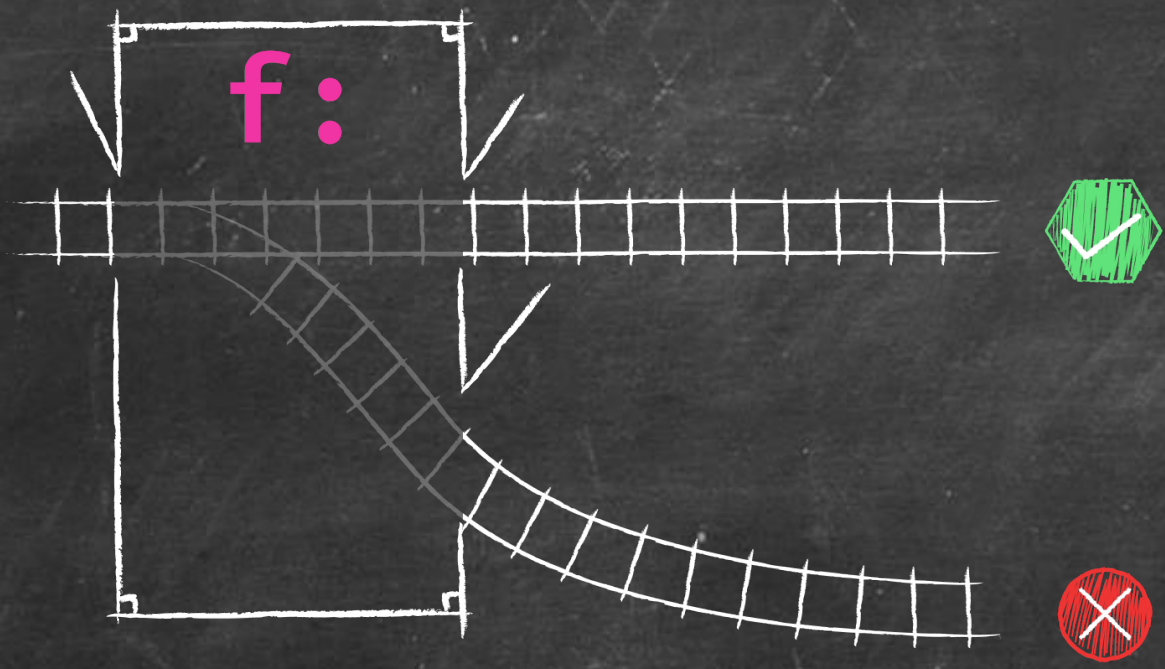
The suspended program

Or in other words, our pure logics.

- Leverage data types to **raise concerns over the data**.
- `Either<L, R>` will be our friend 🤗

Railway oriented programming

Programs as a composition of functions that **can succeed or fail**.


f:

Railway oriented programming

By **Scott Wlaschin** from 11 May 2013

- 📝 Post 👇
  fsharpforfunandprofit.com/posts/recipe-part2/
- 🎬 Talk video + slides 👇
  fsharpforfunandprofit.com/rop/

`Either<L, R>`

A path we want to follow, vs an "alternative" one

- Compute over the happy path 👉 plug error handlers.
- Make disjunction explicit 👉 both paths need to be handled.
- Makes our program complete.

In code 

`Either<A, B>` models this scenario.

```
sealed class Either<out A, out B> {
  data class Left<out A>(val a: A) : Either<A, Nothing>()
  data class Right<out B>(val b: B) : Either<Nothing, B>()

  // operations like map, flatMap, fold, mapLeft...
}
```

- Convention: Errors on `Left`, success on the `Right`.
- Biased towards the `Right` side 👉 compute over the happy path.

fold to handle both sides

```
fun loadUser: Either<UserNotFound, User> =
  Right(User("John")) // or Left(UserNotFound)

// Alternatively: user.right() or exception.left()

val user: Either<UserNotFound, User> =
  loadUser().fold(
    ifLeft = { e -> handleError(e) },
    ifRight = { user -> render(user) }
  )
```

Nullable data

~~`Option<A>`~~ getting deprecated.

- 💡 Alternative 1: `A?`
- 💡 Alternative 2: `Either<Unit, A>`

```
typealias EpisodeNotFound = Unit

fun EpisodeDB.loadEpisode(episodeId: String): Either<EpisodeNo
  Either.fromNullable(loadEpisode("id1"))
    .map { episode -> episode.characters }
```

Integration with effects

Either.catch for 3rd party calls.

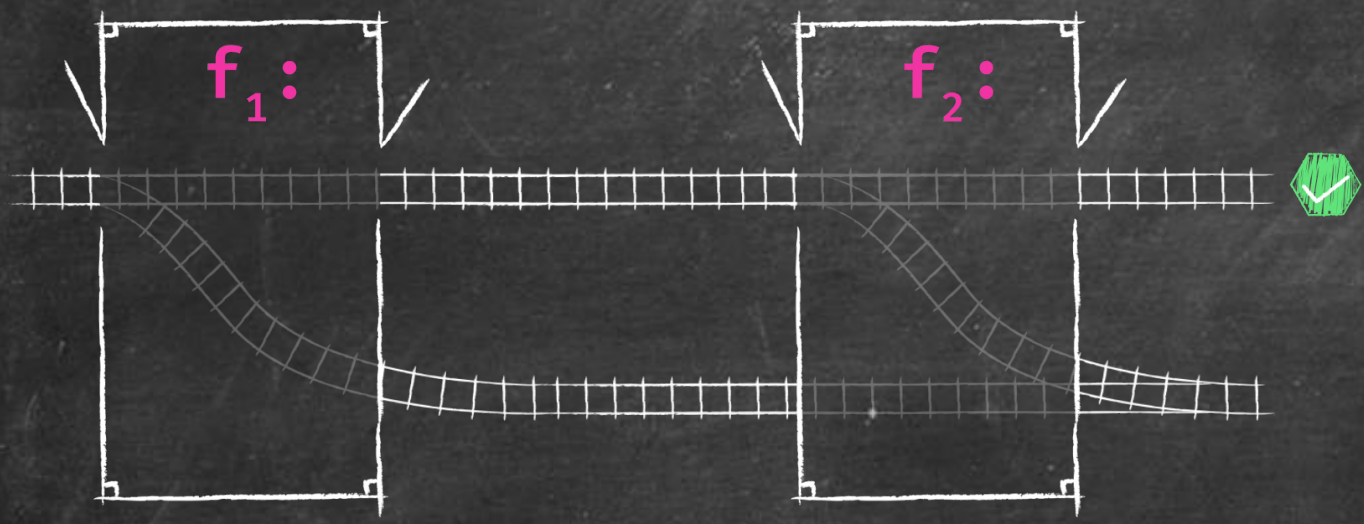
```
suspend fun loadSpeakers(): Either<Errors, List<Speaker>> =
  Either.catch { service.loadSpeakers() } // any suspended op
    .mapLeft { it.toDomainError() } // strongly type errors
```

- Combined with `mapLeft` to map the `Throwable` into something else.
- This program could **never run outside of a controlled environment** 🌟

Composing logics

- We got means to write our logic as pure functions.
- We need the glue for them 👉 `flatMap`.
- Any program 👉 sequence of computations.

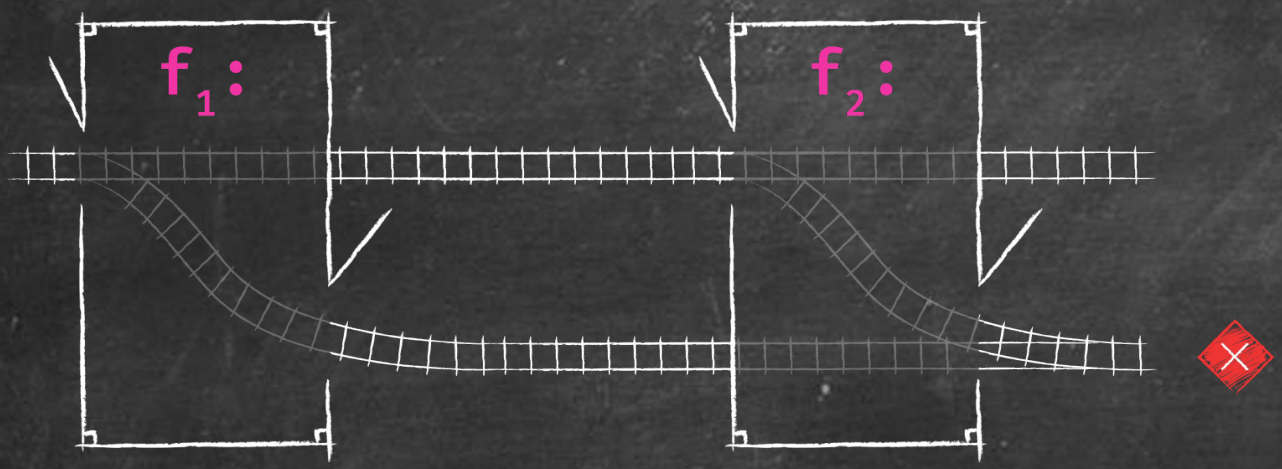
How does `flatMap` work for `Either`?


f_1:
f_2:

Failing fast

When two operations are dependent, you **cannot perform the second one without a successful result by the first one**.

- This means we can save computation time in that case.
- `Either#flatMap` 👉 sequential computations that return `Either`.


f_1:
f_2:

Sequential effects

Here's a program with 2 dependent operations.

```
suspend fun loadSpeaker(id: SpeakerId): Either<SpeakerNotFound,
  TODO()

suspend fun loadTalks(ids: List<TalkId>): Either<InvalidIds, Li
  TODO()

suspend fun main() {
  val talks = loadSpeaker("SomeId")
                .flatMap { loadTalks(it.talkIds) }

  // listOf(Talk(...), Talk(...), Talk(...))
}
```

Sequential effects - bindings

Alternative syntax 👉 Either bindings 👉 sugar 🍬

```
suspend fun main() {
  val talks = either  { // Either<Error, List<Talk>>
    val speaker = !loadSpeaker("SomeId")
    val talks = !loadTalks(speaker.talkIds)
    talks
  }

  // listOf(Talk(...), Talk(...), Talk(...))
}
//sampleEnd
```

Fail fast

First operation fails 👉 short circuits

```
suspend fun main() {
  val events = either {
    val speaker = !loadSpeaker("SomeId") // Left(SpeakerNotFou
    val talks = !loadTalks(speaker.talkIds)
    val events = talks.map { !loadEvent(it.event) }
    events
  }

  println(events) // Left(SpeakerNotFound)
}
```

Error accumulation?

- Interested in **all errors occurring**, not a single one.
- Only in the context of **independent computations**.

Validated & ValidatedNel

```
sealed class Validated<out E, out A> {
  data class Valid<out A>(val a: A) : Validated<Nothing, A>()
  data class Invalid<out E>(val e: E) : Validated<E, Nothing>()
}

typealias ValidatedNel<E, A> = Validated<NonEmptyList<E>, A>
```

- `ValidatedNel` alias for error accumulation on a `NonEmptyList`.

Validated & ValidatedNel

Independent data validation with the applicative.

```
suspend fun loadSpeaker(id: SpeakerId): ValidatedNel<SpeakerNot
  Validated.catchNel { throw Exception("Boom 💥!!") }.mapLeft {

suspend fun loadEvents(ids: List<TalkId>): ValidatedNel<Invalid
  Validated.catchNel { throw Exception("Boom 💥!!") }.mapLeft {

suspend fun main() {
  val accumulator = NonEmptyList.semigroup<Error>()

  val res = Validated.applicative(accumulator)
    .tupledN(
      loadSpeaker("SomeId"),
      loadEvents(listOf("1", "2"))
```

Limitations

- `Either` or `Validated` are **eager**.
- We want them deferred 👉 declarative.
- `suspend` will do the work 👍

But what about **threading / concurrency**?

Arrow Fx Coroutines 🙌

 Arrow Fx Coroutines

- Functional concurrency framework.
- Functional operators to run suspended effects.
- Cancellation system ✅ 👉 All Arrow Fx operators **automatically check for cancellation**.

Environment

Our **runtime**. Picks the execution strategy.

```
// synchronous
env.unsafeRunSync { greet() }

// asynchronous
env.unsafeRunAsync(
  fa = { greet() },
  e = { e -> println(e)},
  a = { a -> println(a) }
)

// cancellable asynchronous
val disposable = env.unsafeRunAsyncCancellable(
  fa = { greet() },
```

- `interface` to implement custom ones.

evalOn(ctx)

Offload an effect to an arbitrary context and **get back to the original one**.

```
suspend fun loadTalks(ids: List<TalkId>): Either<Error.TalksNo
  evalOn(IOPool) { // supports any suspended effects
      Either.catch { fetchTalksFromNetwork() }
          .mapLeft { Error.TalksNotFound }
  }
```

parMapN

- Run N parallel effects.
- Cancel parent 👉 cancels all children.
- Child failure 👉 cancels other children 😲
- All results are required.

```
suspend fun loadEvent(): Event {
    val op1 = suspend { loadSpeakers() }
    val op2 = suspend { loadRooms() }
    val op3 = suspend { loadVenues() }

    return parMapN(op1, op2, op3) { speakers, rooms, venues ->
        Event(speakers, rooms, venues)
    }
}
```

parTupledN

- Same without callback style.
- Returns a tuple with all the results.
- Cancellation works the same way.

```
suspend fun loadEvent(): Event {
    val op1 = suspend { loadSpeakers() }
    val op2 = suspend { loadRooms() }
    val op3 = suspend { loadVenues() }

    val res: Triple<List<Speaker>, List<Room>, List<Venue>> =
        parTupledN(op1, op2, op3)

    return Event(res.first, res.second, res.third)
}
```

parTraverse

- Traverses a dynamic amount of elements running an effect for each, all of them **in parallel**.
- Cancellation works the same way.

```
suspend fun loadEvents() {
    val eventIds = listOf(1, 2, 3)

    return eventIds.parTraverse(IOPool) { id ->
      eventService.loadEvent(id)
    }
}
```

parSequence

- Traverse list of effects, run all in parallel.
- Cancellation works the same.

```
suspend fun main() {
    val ops = listOf(
        suspend { service.loadTalks(eventId1) },
        suspend { service.loadTalks(eventId2) },
        suspend { service.loadTalks(eventId3) })

    ops.parSequence()
}
```

raceN

- Racing parallel effects.
- Returns the winner, cancels losers.
- Cancelling parent 👉 cancels all children.
- Child failure 👉 cancels other children.

```
suspend fun main() {
    val res = raceN(::op1, ::op2, ::op3) // suspended ops
    res.fold(
      ifA = {},
      ifB = {},
      ifC = {}
    )
}
```

Android use case

Racing against the Android lifecycle 🏎️🏁

```
suspend fun AppCompatActivity.suspendUntilDestroy() =
  suspendCoroutine<Unit> { cont ->
    val lifecycleObserver = object : LifecycleObserver {
      @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
      fun destroyListener() {
        cont.resumeWith(Result.success(Unit))
      }
    }
    this.lifecycle.addObserver(lifecycleObserver)
  }

suspend fun longRunningComputation(): Int = evalOn(IOPool) {
  delay(5000)
```

Retrying / repeating

Highly composable retry policies for suspended effects.

```
fun <A> complexPolicy() =
    Schedule.exponential<A>(10.milliseconds)
        .whileOutput { it.seconds < 60.seconds }
        .andThen(spaced<A>(60.seconds) and recurs(100))

suspend fun loadTalk(id: TalkId): List<Talks> =
    retry(complexPolicy()) {
        fetchTalk(id) // retry any suspended effect
    }
```

Concurrent Error handling

All Arrow Fx Coroutines operators rethrow on failure.

- Can use `Either.catch`, `Validated.catch`, `Validated.catchNel`, at any level ✨

And FRP?

Android apps as a **combination of Streams**.

- Inversion of control is strong 👉 Streams can bring determinism to it.
- Unidirectional data flow architectures.
- Lifecycle events, user interactions, application state updates...

What we need

- Emit multiple times.
- **Embed suspended effects.**
- **Compatible with all the Arrow Fx Coroutines operators**.
- Cold streams 👉 purity 👉 declarative.
- Composition.

💡 Pull based Stream

- vs push based alternatives (RxJava, Reactor)
- Receiver `suspends` until data can be pulled.
- Built in back-pressure 🌟

Embedding effects

Evaluates a suspended effect, **emits result**.

```
val s = Stream.effect { println("Run!") }
  .flatMap {}
  .map {}
  ...

s.drain() // consume stream

// Run!
```

- **Cold**. Describes what will happen when the stream is interpreted.
- Terminal operator to run it.
- Errors **raised into the Stream**.

Embedding effects

Any Arrow Fx Coroutines operators can be evaluated.

- Result is emitted over the `Stream`.

```
val s = Stream.effect { // any suspended effect
  parMapN(op1, op2, op3) { speakers, rooms, venues ->
    Event(speakers, rooms, venues)
  }
}

s.drain()
```

- Threading via Arrow Fx Coroutines: `parMapN`, `parTupledN`, `evalOn`, `parTraverse`, `parSequence`... etc.

parJoin

Composing streams **in parallel**.

```
val s1 = Stream.effect { 1 }
val s2 = Stream.effect { 2 }
val s3 = Stream.effect { 3 }

val program = Stream(s1, s2, s3).parJoinUnbounded()
// or parJoin(maxOpen = 3)

val res = program.toList()
println(res)

// [2, 1, 3]
```

- Concurrently 👉 emits values as they come 👉 unexpected order.

async wrapper

Wrap callback based apis.

```
fun SwipeRefreshLayout.refreshes(): Stream<Unit> =
    Stream.callback {
      val listener = OnRefreshListener {
        emit(Unit)
      }
      this@refreshes.setOnRefreshListener(listener)
    }
```

Cancellable async wrapper

Wrap callback based apis in a **cancellable** Stream.

- Return a `CancelToken` to release and avoid leaks.

```
fun SwipeRefreshLayout.refreshes(): Stream<Unit> =
    Stream.cancellable {
      val listener = OnRefreshListener {
        emit(Unit)
      }
      this@refreshes.setOnRefreshListener(listener)

      // Return a cancellation token
      CancelToken { this@refreshes.removeListener(listener) }
    }
```

bracket

Scope resources to the Stream life span.

- Calls release lambda once the Stream terminates.

```
Stream.bracket({ openFile() }, { closeFile() })
    .effectMap { canWorkWithFile() }
    .handleErrorWith { alternativeResult() }
    .drain()
```

Other relevant operators

The usual ones.

```
Stream.effect { loadSpeakers() }
  .handleErrorWith { Stream.empty() }
  .effectMap { loadTalks(it.map { it.id }) } // flatMap + effec
  .map { talks -> talks.map { it.id } }
  .drain() // terminal – suspend
```

- Stays declarative and deferred until `drain()`

interruptWhen + lifecycle

Arbitrary Stream interruption by **racing streams**.

- Will terminate your program as soon as a lifecycle `ON_DESTROY` event is emited.

```
program()
    .interruptWhen(lifecycleDestroy()) // races both
    .drain()
```

interruptWhen + lifecycle

Stream out of lifecycle events 👉 destroy

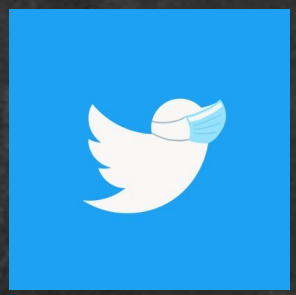
```
fun Fragment.lifecycleDestroy(): Stream<Boolean> =
    Stream.callback {
      viewLifecycleOwner.lifecycle.addObserver(
          LifecycleEventObserver { _, event ->
            if (event == Lifecycle.Event.ON_DESTROY) {
              emit(true)
            }
          })
    }
```

Consuming streams safely

Terminal ops are suspend 👉 **Stream has to run within a safe environment**.

```
class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    val env = Environment()
    env.unsafeRunAsync {
      HomeDependencies.program() // program as a Stream
        .interruptWhen(lifecycleDestroy())
        .drain() // suspended – terminal op to consume
    }
  }
}
```

Thank you! 🙌



@JorgeCastilloPr

To expand on these ideas 👉 Fully-fledged **Functional Android course**.

- www.47deg.com/trainings/Functional-Android-development/
- Bookable as a group / company.