How functional is direct-style?

Adam Warski
warski.org

SOFTWAREMILL

Part I

What is Functional Programming?

Functional Programming:

a paradigm where programs are constructed by applying and composing functions

Wikipedia

Traits of "functional" approach	Non-traits of "functional" style
Higher-order functions	Loops
Functions as first-class values	Code duplication
Expressions	Statements
Immutable data	Mutable data
Effects as values	Immediate effects
Function composition	Imperative steps
Data & behaviour separate	Data & behaviour combined

What is a function?

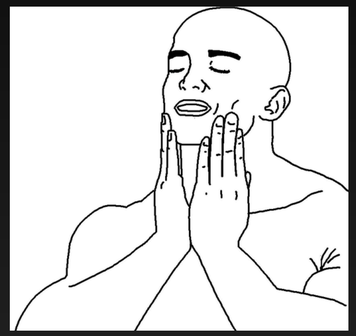
"casual" FP

- a callable unit of software
- well-defined interface & behaviour
- can be invoked multiple times

"pure" FP

- $f: D \to C$, for each $x \in D$, exactly one $f(x) \in C$
- idempotent
- effect-free

Functionfullness



% of FP

Spaghetti code

Clean code (small methods)

Casual FP

Pure FP

\+ higher-order functions
\+ ADTs, immutable data
\+ expression-oriented
\+ ...

```
var users = new ArrayList[User]()
for id <- peopleIds do
  users.add(fetchFromDb(id))

var likes = new ArrayList[Response]()
var dislikes = new ArrayList[Response]()
for user <- users do
  val response = sendRequest(user)
  if response.body.likedSciFiMovies
    .contains("Star Wars")
  then likes.add(response)
  else dislikes.add(response)
```

```
peopleIds
  .map(fetchFromDb)
  .map(sendRequest)
  .map(_.body.likedSciFiMovies)
  .partition(_.body
    .likedSciFiMovies
    .contains("Star Wars"))
```

```
peopleIds
  .traverse: id =>
    fetchFromDb(id)
      .flatMap(sendRequest)
  .map(_.partition(_.body
    .likedSciFiMovies
    .contains("Star Wars")))
```

What is FP?

Casual FP	Pure FP
Composing functions	
Light syntax for lambdas	
Higher-order functions	🍀 No mutation
Immutable data types	🍀 No side effects
ADTs + pattern matching	🍀 All computations are lazy
Expression-oriented	
Functional standard library	
Separate data & behavior	

26

> What is functional programming

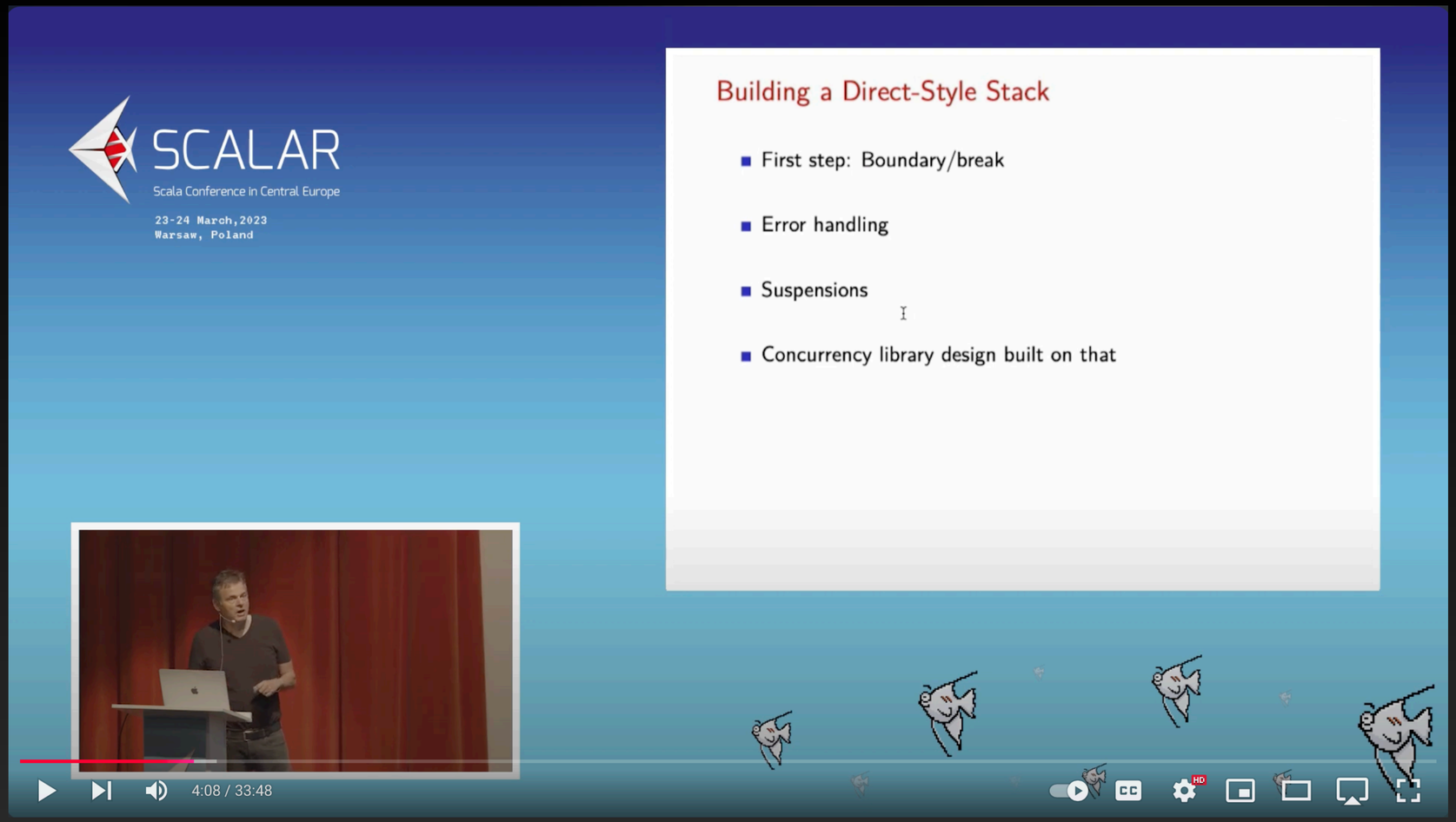
There are two different definitions of "functional programming" in common use today:

The older definition (originating from Lisp) is that functional programming is about programming using first-class functions, i.e. where functions are treated like any other value so you can pass functions as arguments to other functions and function can return functions among their return values. This culminates in the use of higher-order functions such as `map` and `reduce` (you may have heard of `mapReduce` as a single operation used heavily by Google and, unsurprisingly, it is a close relative!). The .NET types `System.Func` and `System.Action` make higher-order functions available in C#. Although currying is impractical in C#, functions that accept other functions as arguments are common, e.g. the `Parallel.For` function.

The younger definition (popularized by Haskell) is that functional programming is also about minimizing and controlling side effects including mutation, i.e. writing programs that solve problems by composing expressions. This is more commonly called "purely functional programming". This is made possible by wildly different approaches to data structures called "purely functional data structures". One problem is that translating traditional imperative algorithms to use purely functional data structures typically makes performance 10x worse. Haskell is the only surviving purely functional programming language but the concepts have crept into mainstream programming with libraries like `Linq` on .NET.

Part II

What is Direct Style?



- Direct style is the opposite of continuation-passing style and a control monad

Direct-style

- Results of effectful computations are available directly
- Not wrapped with a `Future`, `Promise`, `IO` or `Task`
  - (by default)
- Avoiding continuations
  - as callbacks
  - as monadic wrappers

But!

We need continuations for performance

- How to: continuations + direct syntax?
  - Kotlin's coroutines
  - Java's VirtualThreads
  - Unison Abilities
  - OCaml's EIO

```
let send_response socket =
  Eio.Flow.copy_string "HTTP/1.1 200 OK\r\n" socket;
  Eio.Flow.copy_string "\r\n" socket;
  Fiber.yield ();       (* Simulate delayed generation of body *)
  Eio.Flow.copy_string "Body data" socket
```

```
# Eio_main.run @@ fun _ ->
  send_response (Eio_mock.Flow.make "socket");;
+socket: wrote "HTTP/1.1 200 OK\r\n"
+socket: wrote "\r\n"
+socket: wrote "Body data"
- : unit = ()
```

```
safeDiv3 : '{IO, Exception, Store Text} Nat
safeDiv3 =
  do
    use Nat / == toText
    use Text ++
    a = randomNat()
    b = randomNat()
    Store.put (toText a ++ "/" ++ toText b)
    if b == 0 then Exception.raise (Generic.failure "Oops. Zero" b)
    else a / b
```

Part III

How functional is direct style?

Scala is functional by construction

- functions as 1st class values
- higher-order functions
- expression-oriented
- immutable data types / ADTs

We only need to avoid spoiling what we got

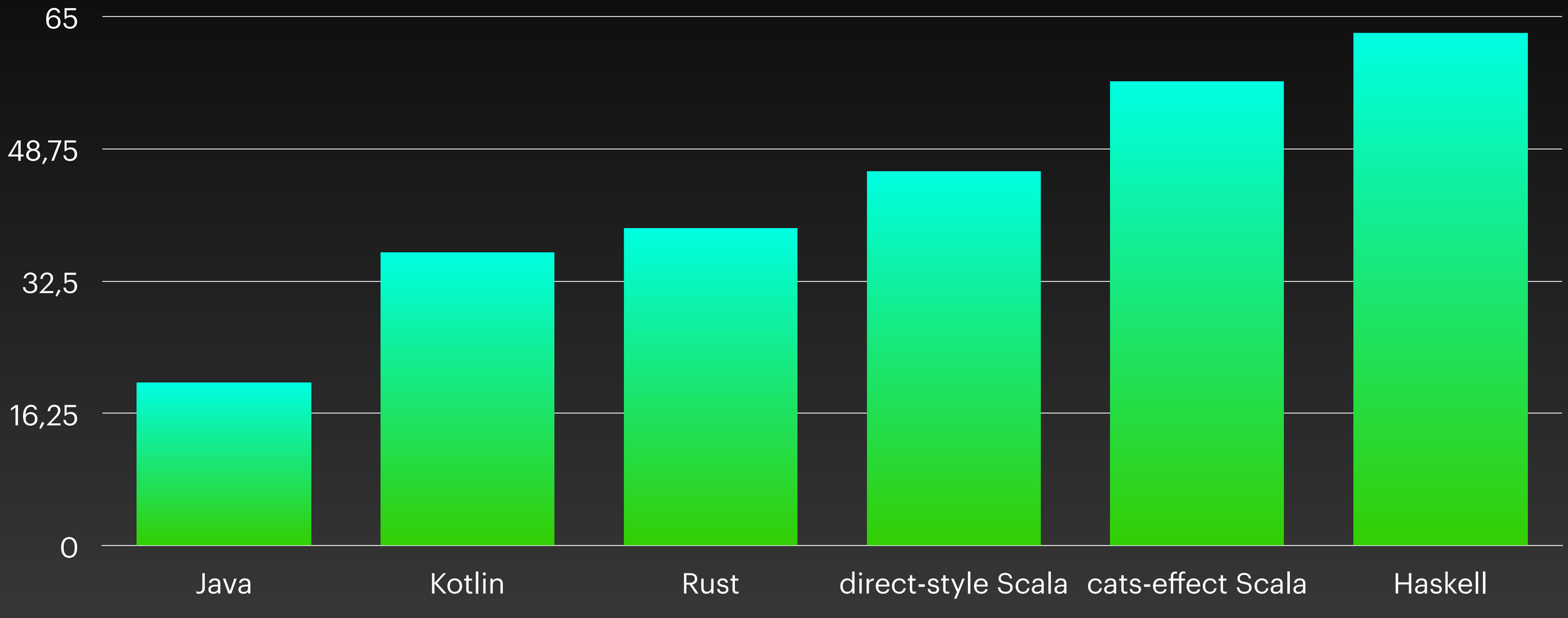
Functionality scorecard

	Scala + cats-effect / ZIO	
Functions as 1st-class values	10 / 10	Scala
Expression-oriented	4 / 5	
Functional std lib	5 / 5	
Applying & composing functions	9 / 10	based on discipline
Immutable data, ADTs	9 / 10	
No shared mutable state	9 / 10	
Effects as values / no effects	8 / 10	
Behaviour & data separate	3 / 5	OO / FP hybrid
	57 / 65	

Functionality scorecard

	Direct-style	
Functions as 1st-class values	10 / 10	Scala
Expression-oriented	4 / 5	
Functional std lib	5 / 5	
Applying & composing functions	6 / 10	based on discipline
Immutable data, ADTs	8 / 10	
No shared mutable state	8 / 10	
Effects as values / no effects	2 / 10	Errors as `Either`s
Behaviour & data separate	3 / 5	OO / FP hybrid
	46 / 65	

Functionality scorecard



Is direct-style at odds with pure FP (in Scala)?

- Limited direct-style is possible
  - zio-direct
  - `async`/`await` for cats-effect
- Higher-order functions problematic
- Unlimited direct-style with pure, monadic FP seems impossible

```
defer {
  val textA = read(fileA).run
  if (fileA.contains("some string")) {
    val textB = read(fileB).run
    write(fileC, textA + textB).run
  }
}
```

```
import cats.effect.IO
import cats.effect.cps._

val program: IO[Int] = async[IO] {
  var n = 0
  var i = 1

  while (i <= 3) {
    n += IO.pure(i).await
    i += 1
  }

  n
}
```

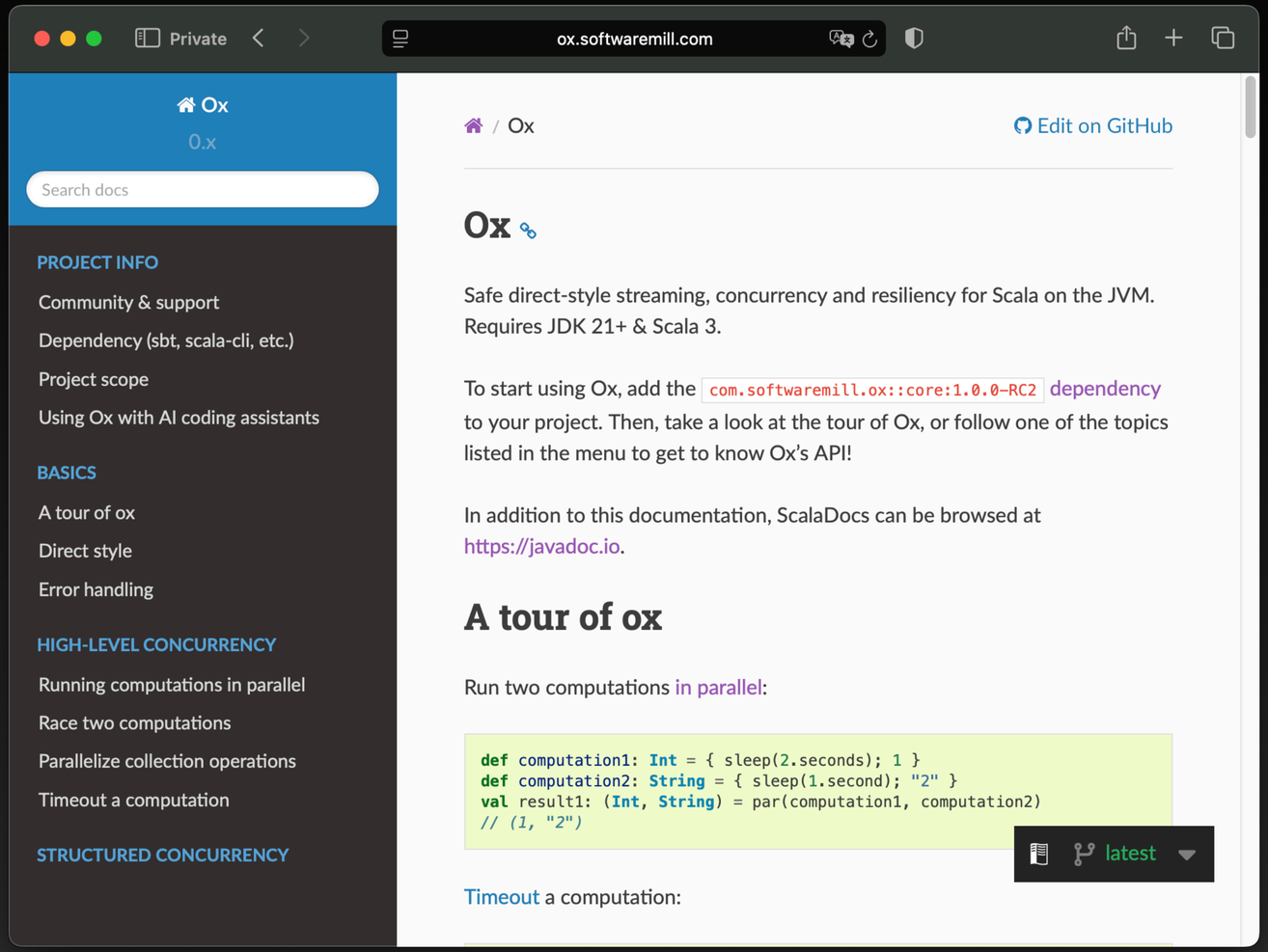
There's more to functional effects

than fibers

The good	The bad
🍀 Error handling	💔 Syntax overhead
🍀 Resource management	💔 Custom control flow
🍀 Fearless refactoring	💔 Lost error context
🍀 Principled interruptions	💔 Virality
🍀 High-level concurrency	💔 Learning curve
🍀 Streaming	

Can we ...

- Use casual FP
- Leverage Java 21's virtual threads
- Keep some of the benefits of purely-functional effect systems
- But avoid some of the problems


Private
ox.softwaremill.com
Ox
0.x
Search docs
PROJECT INFO
Community & support
Dependency (sbt, scala-cli, etc.)
Project scope
Using Ox with AI coding assistants
BASICS
A tour of ox
Direct style
Error handling
HIGH-LEVEL CONCURRENCY
Running computations in parallel
Race two computations
Parallelize collection operations
Timeout a computation
STRUCTURED CONCURRENCY
Ox
Edit on GitHub
Ox
Safe direct-style streaming, concurrency and resiliency for Scala on the JVM. Requires JDK 21+ & Scala 3.
To start using Ox, add the com.softwaremill.ox::core:1.0.0-RC2 dependency to your project. Then, take a look at the tour of Ox, or follow one of the topics listed in the menu to get to know Ox's API!
In addition to this documentation, ScalaDocs can be browsed at https://javadoc.io.
A tour of ox
Run two computations in parallel:
def computation1: Int = { sleep(2.seconds); 1 }
def computation2: String = { sleep(1.second); "2" }
val result1: (Int, String) = par(computation1, computation2)
// (1, "2")
latest
Timeout a computation:

Ox docs

```
def computation1: Int = { sleep(2.seconds); 1 }
def computation2: String = { sleep(1.second); "2" }
val result1: (Int, String) = par(computation1, computation2)
// (1, "2")
```

```
supervised:
  val f1 = fork { sleep(2.seconds); 1 }
  val f2 = fork { sleep(1.second); 2 }
  (f1.join(), f2.join())
```

```
useCloseable(new java.io.PrintWriter("test.txt")) { writer =>
  writer.println("Hello, world!")
}
```

```
Flow.iterate(0)(_ + 1) // natural numbers
  .filter(_ % 2 == 0)
  .map(_ + 1)
  .intersperse(5)
  // compute the running total
  .mapStateful(0) { (state, value) =>
    val newState = state + value
    (newState, newState)
  }
  .take(10)
  .runForeach(n => println(n.toString))
```

The error model

- Different constructs for recoverable errors & programming bugs
- Indication at callsite, where errors might occur
- Recoverable errors part of the type

```scala
def lookupUser(id1: Int): Either[String, User] = ???
def lookupOrganization(id2: Int): Either[String, Organization] = ???

val result: Either[String, Assignment] = either:
  val user = lookupUser(1).ok()
  val org = lookupOrganization(2).ok()
  Assignment(user, org)
```

Part IV

Concluding ...

Direct-style Scala is "casually" functional

(and it’s practical to use!)

Comparing Ox to functional effect systems

The better	The same	The worse
🍀 Casual FP	⭐ Fearless concurrency	💔 Pure FP
🍀 Simple syntax	⭐ Streaming	💔 Principled error handling
🍀 Lower learning curve	⭐ Structured concurrency	💔 Principled interruptions
🍀 Exceptions retain context		💔 Dedicated resource type
🍀 No virality		💔 Uniform computations
🍀 Built-in control flow		💔 Fearless refactoring

Links

- Wikipedia on FP
- Why FP Matters by John Hughes
- Why FP Doesn’t Matter (Jane Street)
- What’s FP? @ StackOverflow
- What’s FP @ Haskell
- Direct Style Scala by Martin Odersky @ Scalar
- Notes on structured concurrency by Nathaniel J. Smith
- The error model by Joe Duffy

Deep dives

- What is FP @ Lambda Days (slides, video)
- Unwrapping IO @ Scala.IO (slides, video)
- From Reactive Streams, to Virtual Threads

IO.pure("Thank you!")

warski.org