How functional is direct-style?

Adam Warski, November 2024
warski.org

SOFTWAREMILL

Part I

What is Functional Programming?

A practical take

"You know it when you see it"

Function application

```
const numbers = [1, 2, 3, 4, 5, 6];

const result = numbers
  .map((num) => num * 2)
  .filter((num) => num % 2 === 0)
  .reduce((acc, num) => acc + num, 0);

console.log(result);
```

Higher-order function

Loop

```
const numbers = [1, 2, 3, 4, 5, 6];

let result = 0;

for (let i = 0; i < numbers.length; i++) {
  const doubled = numbers[i] * 2;
  if (doubled % 2 === 0) {
    result += doubled;
  }
}

console.log(result);
```

Code duplication

```
var menu: List[Meal] = Nil

if vegetarian then
  menu = entireMenu.filter(_.ingredients.exists(_.meat))
else
  menu = entireMenu.filter(_.ingredients.forall(_.plant))
```

Function as first-class values

```
val dietaryRestriction: Meal => Boolean = if vegetarian
  then m => m.ingredients.exists(_.meat)
  else m => m.ingredients.forall(_.plant)

val menu = entireMenu.filter(dietaryRestriction)
```

`if` as an expression

```scala
case class Person(name: String, age: Int)
case class Address(where: String, permanent: Boolean)

val chris = Person("Chris", 29)
val friends = Map(
  chris -> Address("Warsaw", permanent = true),
  Person("Nicole", 36) -> Address("Paris", permanent = false)
)

val updatedFriends = friends
  .removed(chris)
  .updated(chris.copy(age = 30),
    Address("Kraków", permanent = true))
```

Immutable data

```java
class Person { String name; int age; }
class Address { String where; boolean permanent; }

Person chris = new Person("Chris", 29);
Map<Person, Address> friends = new HashMap<>();
friends.put(chris, new Address("Warsaw", true));
friends.put(new Person("Nicole", 36),
  new Address("Paris", false));

chris.setAge(30);
friends.put(chris, new Address("Kraków", true));
```

Mutable data

```java
public class RocketLauncher {
    public void launch() {
        checkFuelLevel();
        initializeNavigation();
        performSystemCheck();
        igniteEngine();
        System.out.println("Success!");
    }
}
```

Immediate effects

```haskell
launchRocket :: IO ()
launchRocket = do
  checkFuelLevel
  initializeNavigation
  performSystemCheck
  igniteEngine
  putStrLn "Success!"
```

Effects as values

```haskell
launchRocket : '{IO}
launchRocket _ =
  checkFuelLevel
  initializeNavigation
  performSystemCheck
  igniteEngine
  printLine "Success!"
```

Somewhere in-between

```
val order = Order.parse(request)
order.validate()
order.calculateTotal()
if request.loyalCustomer then
  order.setLoyal(true)
  order.applyDiscount()
order.calculateTax()

new Invoice(order.getTotal(), order.getTax())
```

Imperative steps

```
parseOrder(request)
  .flatMap(validateOrder)
  .map: order =>
    createInvoice(
      applyDiscount(calculateTotal(order), request),
      calculateTax(order))
```

Function composition

Object-oriented vs functional

OO	FP
Combines state & behaviour	Behaviour and data are separate
Interaction vs message passing	Composing functions
State is protected (encapsulation)	State is immutable

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

A theoretical take

Are these traits rooted in theory?

Functional Programming:

a paradigm where programs are constructed by applying and composing functions

Wikipedia

https://en.wikipedia.org/wiki/Functional_programming

Applying & composing functions

- not only being able to, but primary tool used to structure code
- declarative: expressing the logic without control flow
- as a result: trees of expressions
- requires: functions are 1st class values

What is a function?

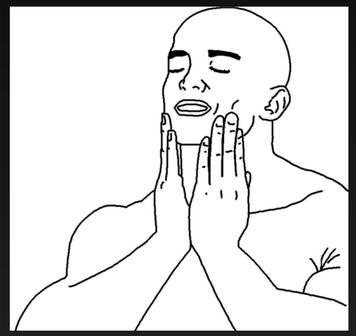
"casual" FP

- 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")))
```

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

erent approaches to data structures called
s that translating traditional imperative
s typically makes performance 10x worse.
ramming language but the concepts have
like `Linq` on .NET.

94

One key feature in a functional language is the concept of first-class functions. The idea is that you can pass functions as parameters to other functions and return them as values.

Functional programming involves writing code that does not change state. The primary reason for doing so is so that successive calls to a function will yield the same result. You can write functional code in any language that supports first-class functions, but there are some languages, like Haskell, which do not allow you to change state. In fact, you're not supposed to make any side effects (like printing out text) at all - which sounds like it could be completely useless.

https://stackoverflow.com/questions/24279/functional-programming-and-non-functional-programming

r/functionalprogramming • 1 yr. ago
Inconstant_Moo

What even is functional programming and why do we like it?

Question

One answer I've got from asking this question is "it's a vibe". And this may be a reasonable answer! (See Wittgenstein's discussion of what a "game" is.) So there's a sort of ... not even a spectrum, but a vague cloud ... which embraces both pure lazy languages and Lisp.

But there might be an actual definition. The nearest I can come up with is that **a functional language is one in which it would be hard or impossible to do ordinary easy things if you didn't use functions as first-class objects**.

Puzzleheaded-Lab-635 • 1y ago •

> A functional language, is a language that prioritizes referential transparency and immutability in its semantics and ergonomics.

https://www.reddit.com/r/functionalprogramming/comments/16ud127/what_even_is_functional_programming_and_why_do_we/

What is Haskell's take?

- Functional programming is a style of programming which models computations as the evaluation of expressions.
- (…) in contrast with imperative programming where programs are composed of statements which change global state when executed.
- Functional programming typically avoids using mutable state.

https://wiki.haskell.org/Functional_programming

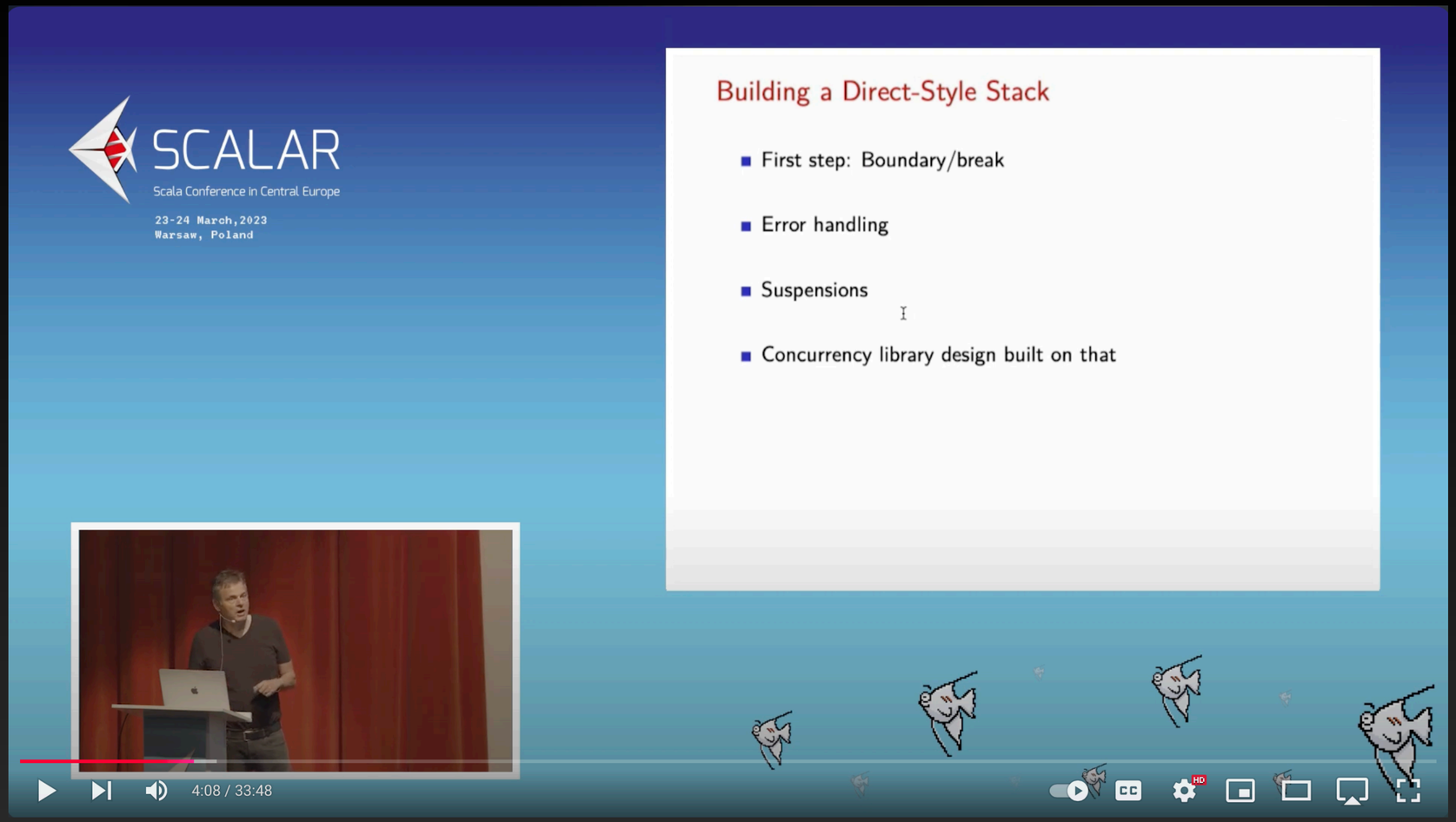
What is functional programming?

- Applying & composing functions
- Functions as 1st-class values
- Expression-oriented
- Immutable data, ADTs
- No shared mutable state
- Effects as values / no effects
- Behaviour & data separate

Part II

What is Direct Style?

No wikipedia entry!



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

https://www.youtube.com/watch?v=0Fm0y4K4YO8

Some key characteristics of direct style programming:

1. **Sequential Execution**: Each statement or function call completes its execution before moving to the next line.
2. **Readable Flow**: Because the code flows in a straightforward manner, it's often more readable and intuitive.
3. **Blocking**: Functions usually "block" until they complete, which can sometimes lead to inefficiencies if waiting on long-running operations like I/O tasks.
4. **Stack-Based Control Flow**: Each function call pushes a frame onto the stack, returning only once the function has finished.

Example Comparison

Consider a simple function that reads a file and processes its data:

- **Direct Style**:

```python
def process_file(filename):
    data = read_file(filename)  # Blocks until the file is read
    return process_data(data)    # Processes the data
```

- **Continuation-Passing Style (CPS)** (a non-direct style):

```python
def process_file(filename, callback):
    read_file_async(filename, lambda data: callback(process_data(data)))
```

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
```

- Imperative often has specific connotations (global state, statements)
- Imperative might be both based on continuations & direct
- Plus:

```
launchRocket :: IO ()
launchRocket = do
  checkFuelLevel
  initializeNavigation
  performSystemCheck
  igniteEngine
  putStrLn "Success!"
```

Direct style consequences

- Direct syntax
  - using the language's built-in control flow constructs
- Direct execution
  - no intermediate library-level runtime
- Direct effects
  - not suspended / as a value

What is direct style?

Results of effectful computations are available directly

Part III

How functional is direct style?

How functional is direct style?

As much, as possible, short of introducing IO

Is direct-style at odds with pure FP?

- Limited direct-style is possible
  - zio-direct
  - `async/await` for cats-effect
- Higher-order functions problematic
- Unlimited direct-style 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
}
```

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 `Eithers`
Behaviour & data separate	3 / 5	OO / FP hybrid
	46 / 65	

The shared mutable state is always there

Thread/fiber 1:

```
writeToDatabase(user.copy(age = user.age + 1))
```

Thread/fiber 2:

```
readFromDatabase(userId)
```

Functionality scorecard

	Java	
Functions as 1st-class values	8 / 10	Java
Expression-oriented	2 / 5	
Functional std lib	0 / 5	
Applying & composing functions	2 / 10	
Immutable data, ADTs	3 / 10	
No shared mutable state	3 / 10	
Effects as values / no effects	0 / 10	
Behaviour & data separate	2 / 5	Data-oriented programming
	20 / 65	

Functionality scorecard

	Kotlin	
Functions as 1st-class values	10 / 10	Kotlin
Expression-oriented	4 / 5	
Functional std lib	2 / 5	
Applying & composing functions	6 / 10	
Immutable data, ADTs	6 / 10	
No shared mutable state	6 / 10	
Effects as values / no effects	0 / 10	
Behaviour & data separate	2 / 5	
	36 / 65	

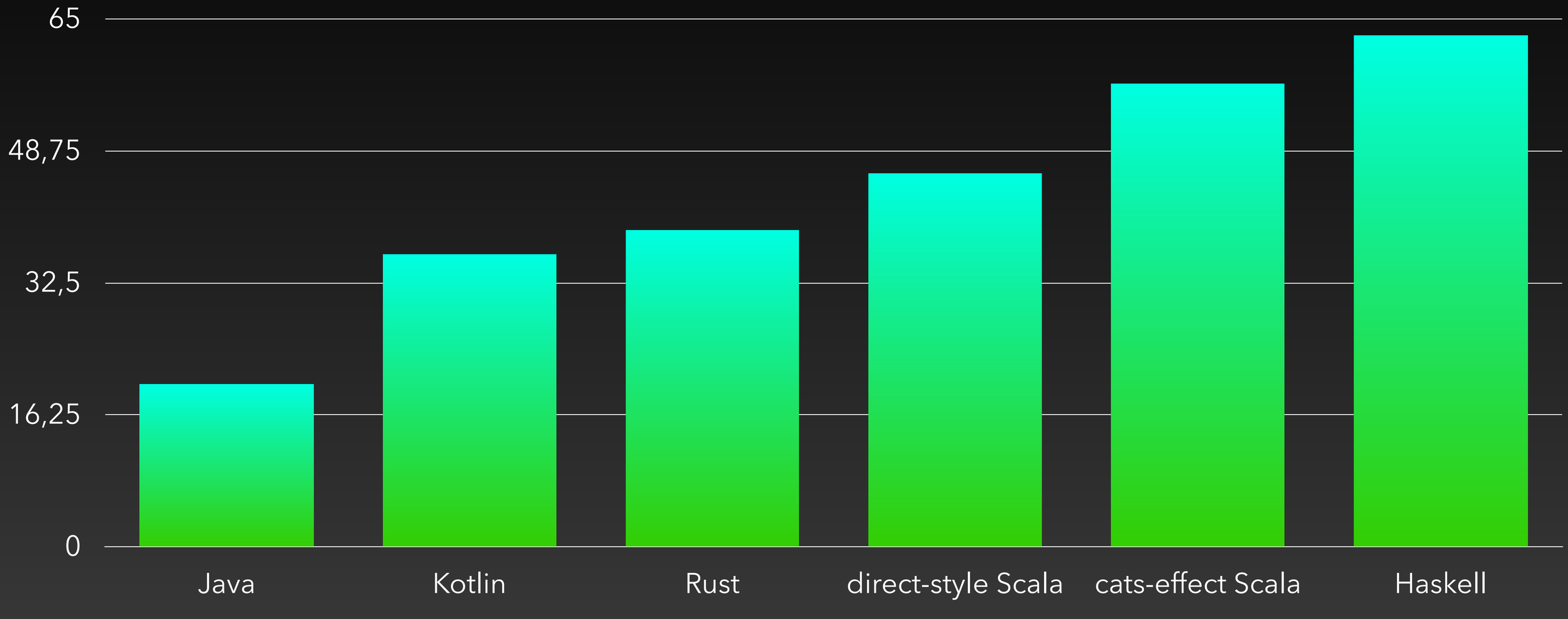
Functionality scorecard

	Haskell	
Functions as 1st-class values	10 / 10	Haskell
Expression-oriented	5 / 5	
Functional std lib	5 / 5	
Applying & composing functions	10 / 10	
Immutable data, ADTs	10 / 10	
No shared mutable state	9 / 10	
Effects as values / no effects	9 / 10	`unsafePerformIO`
Behaviour & data separate	5 / 5	
	63 / 65	

Functionality scorecard

	Rust	
Functions as 1st-class values	10 / 10	Rust
Expression-oriented	4 / 5	
Functional std lib	3 / 5	
Applying & composing functions	6 / 10	
Immutable data, ADTs	3 / 10	
No shared mutable state	9 / 10	`unsafe`
Effects as values / no effects	0 / 10	
Behaviour & data separate	5 / 5	
	39 / 65	

Functionality scorecard



Part IV

Concluding …

How functional is direct style?

Pretty functional

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

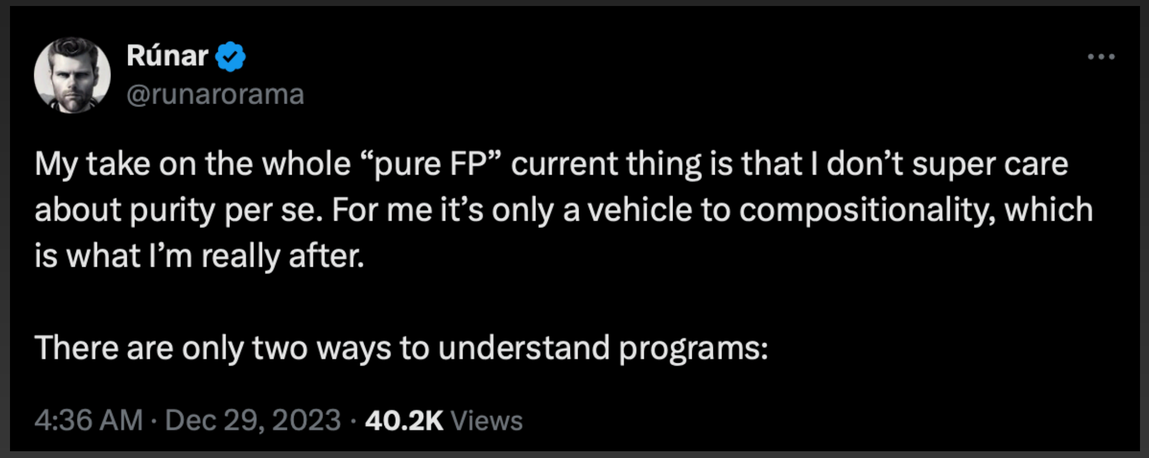
Ox-flavored direct style

- Blocking I/O
- Exceptions for panics, `Either`s (values) for errors
- No shared mutable state
- Using immutable data structures (locally mutable ok)
- Favour function composition
- No shared mutable state
- Channels for inter-thread communication

https://ox.softwaremill.com/latest/

At which point FP provides diminishing returns?

- If you can go purely functional—great
- Most of FP's value is available in the casual variant
- FP in Scala is ergonomic & practical (language, std lib, community)



https://x.com/runarorama/status/1740577439842279821

Solving the hard problems that our clients face using software

IO.pure("Thank you!")

warski.org