What is a (side) effect?

Alejandro Serrano Mena

LambdaConf 2018

Does this code have side effects?

```
[3, 5] ++ [8]
```

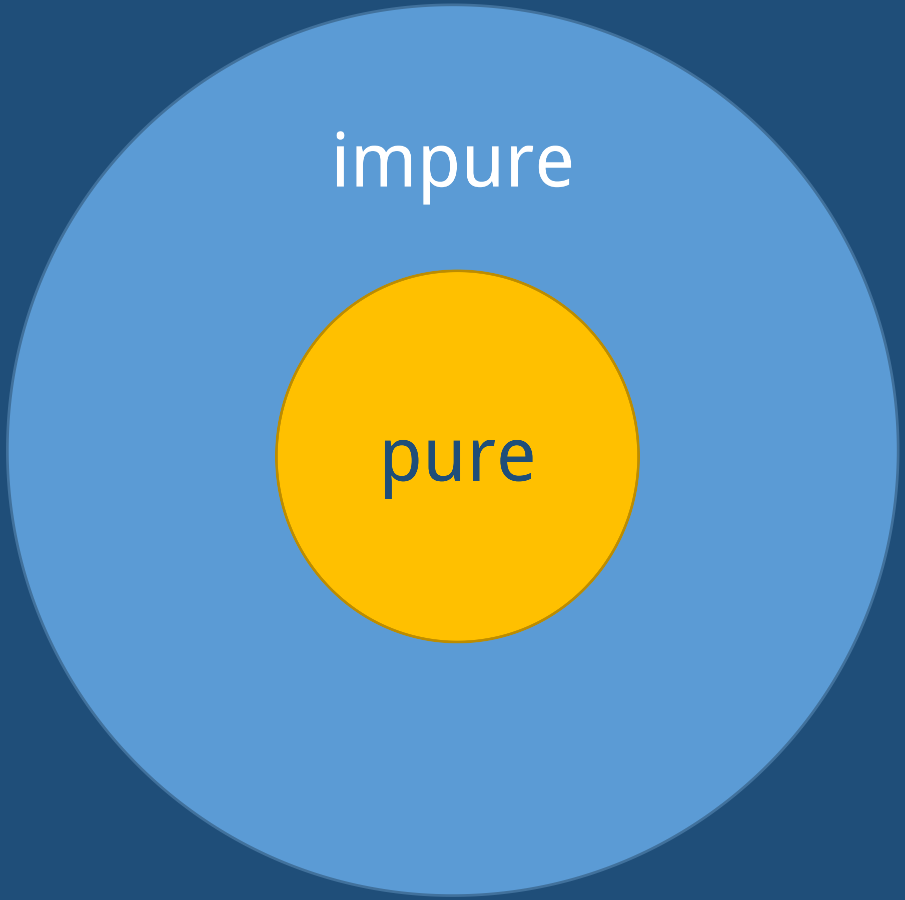
```
List(3,5) ++ List(8)
```

We do not know *yet*

How to *handle* (side) effects

Even what a (side) effect *is*

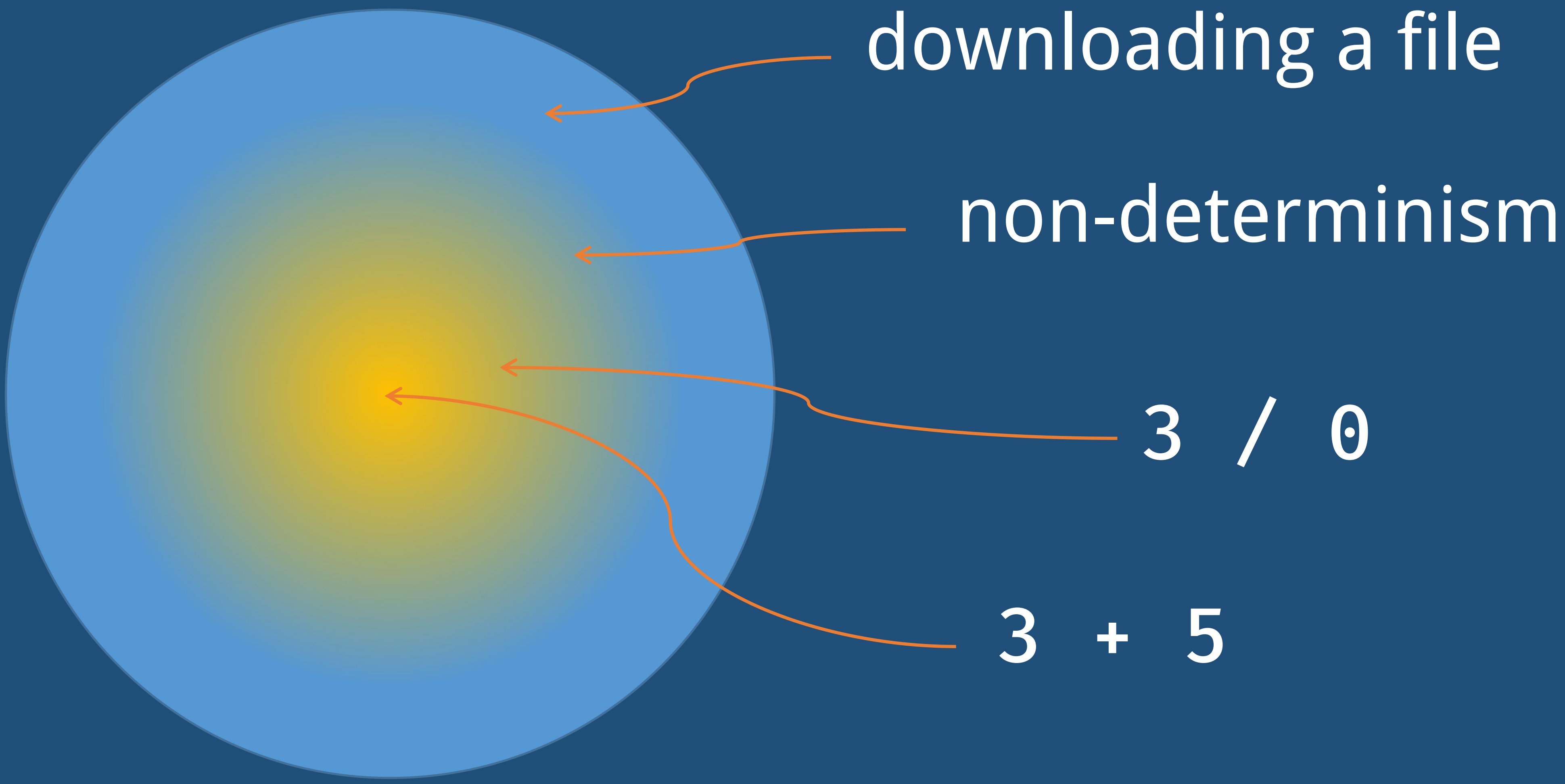
The world for a believer



- Talks to the world
- Difficult to reason about
- Terrible but necessary

- Obeys laws of logic
- We never want to leave

A more realistic view



downloading a file

non-determinism

`3 / 0`

`3 + 5`

Many proposals to handle effects

How do we describe effects in code?

- Monads
- Algebraic handlers / effects
- Type and effect systems

Monads

Insight: effects share a common interface

```
id     :: a ->   a
return :: a -> m a
apply  ::   a -> (a ->   b) ->   b
bind   :: m a -> (a -> m b) -> m b
comp   :: (b ->   c) -> (a ->   b) -> (a ->   c)
(>=>)  :: (b -> m c) -> (a -> m b) -> (a -> m c)
```

Monads

There are so many monads out there!

- Lists / non-determinism
- Errors: Maybe/Option, Either
- State, both pure and with references
- Context and DI: Reader
- Async. computations and promises
- Resource management
- Database access
- …

Too many monads?

Petricek: we are obsessed with monads

FP with strong types community

- Rite of passage
  - "Monad is a mono
- The monad instan
  - Being the "Fluguz monad" is better

How long until we realized that parsers work better as Applicative?

Combining monads

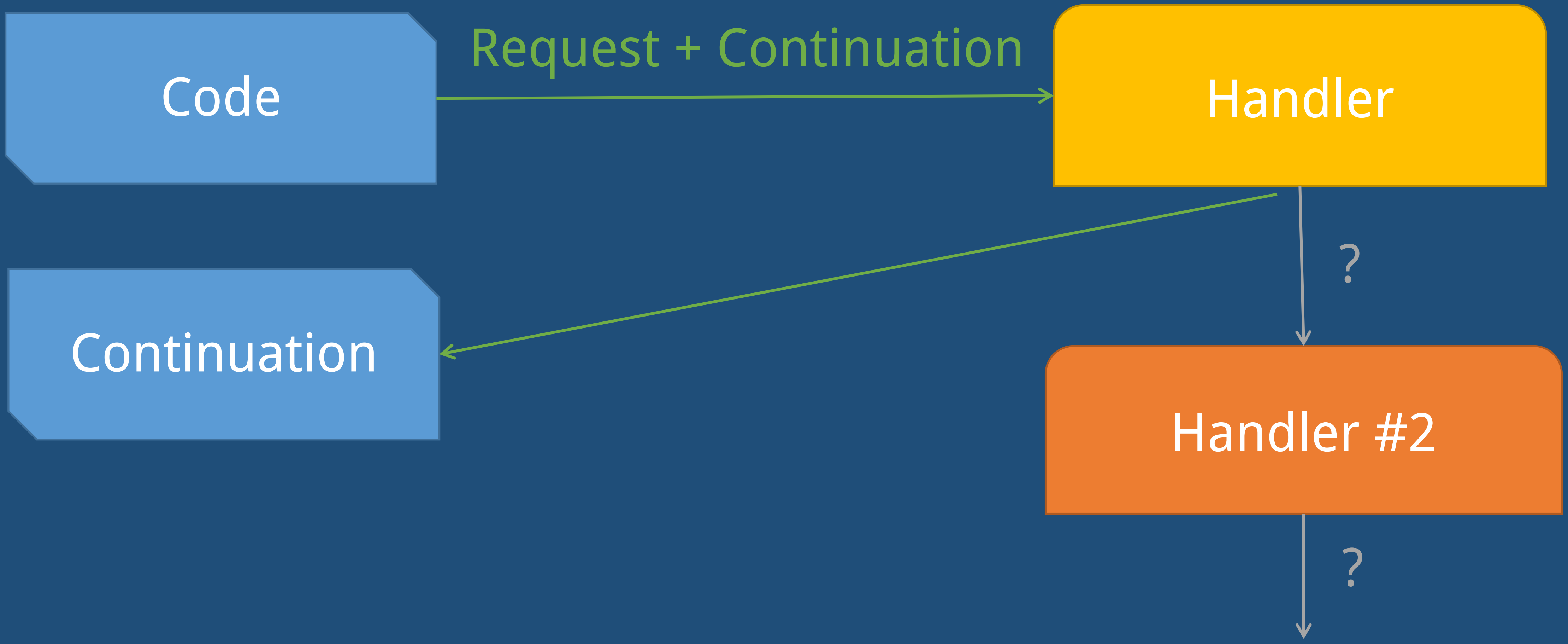
Insight 1: order matters

`Maybe [a]` is different from `[Maybe a]`

Insight 2: some operations are more difficult to combine than others

```
catch :: m a -> (e -> m a) -> m a
```

Algebraic handlers / effects



Code

Request + Continuation

Handler

?

Continuation

Handler #2

?

Algebraic handlers / effects

Effects look like the algebra of a monad

```
data StateA s next a = Get (s -> next a)
                     | Put s (next a)
data MaybeA next a = Error
```

- Easy to combine: join the messages!
- Still, you need to order the handlers

Algebraic handlers / effects

Insight 2':
`throw` is a message, `catch` is a handler

`catch` says how to react to a `throw`

Type and effect systems

Every computation gets
type + description of effects

```
print      s :: ()     ; Console
openDbPool c :: Handle ; Resource, Database
```

```
                    Console
print   :: String  ------> ()   ; None
                     Error
problem :: Something ---> Other ; None
```

Type and effect systems

Insight: effects are about computations, other approaches mix them with values

"A value is, a computation does"

- Paul Levy

Still…

Your effect is not my effect

Reading a value from memory

- In Haskell
  - Not an effect if part of an immutable value
  - An effect from a mutable variable
- In Rust
  - May involve borrowing the variable
- In a language with access control
  - Readin must comply with the policy

How fine-grained?

`IO` is at the same time

- Too wide
  - Nework, file operations, mutable vars...
- Too narrow
  - Look at Scalaz's IO[E,A]

And not even talking about performance...

Lack of a rich language

```
openFile :: FilePath -> IOMode -> IO Handle

Open :  (fname : String)
     -> (m : Mode)
     -> sig FileIO Bool ()
          (\res => case res of
                     True => OpenFile m
                     False => ())
```

Dependent types!

effect = operations + laws

An algebraic approach

The algebraic programme, #1

Categories embody the concept of "composition"

- Do not fixate in monads
- Explore new alternatives
  - Weaker: arrows, applicatives...
  - Stronger: indexed monads...

The algebraic programme, #2

Add laws to describe the behavior of the operations

- "Purification" of the effect
- We like it if we can reason about it

The new landscape

pure	effectful	side-effectful
	few primitives	no simple set of primitives
	lots of laws	no real laws
	non-determinism	*"throw-all"* ***IO***

Data types with laws?

Homotopy Type Theory (HoTT)

- Higher-inductive types

```
data State s a where
  Get : State s s
  Put : s -> State s ()
  PutGetLaw : Get   >>= Put    == Return ()
  PutPutLaw : Put s >>= Put s' == Put s'
```