# Pure Functional Programming

Die letzte Meile

# What is pure FP, and why bother?

- Are these the same?

```
foo("ha") |+| foo("ha")
```

```
val ha = foo("ha")
ha |+| ha
```

- If foo is `_.length`, they are. If foo is `println`, they're not.
- functions like `_.length` are called "referentially transparent"
  - function calls can always be replaced by the values that they calculate
- much easier to reason about
- so let's make every function like that!

# Totality

- Referential transparency implies: every function must be total
  - Must return after a finite time
  - Must not throw exceptions
  - Must not return null

Leads to:

- fewer crashes (invalid arguments ruled out by type system)
- easier refactorings (“if it compiles, it’s probably OK”)

# Determinism

- Passing the same arguments to a function must lead to the same result
- Much easier to test: no need to set up state, just pass some arguments to a function and compare the result
- everything that’s relevant shows up in the function’s signature, thus easier to follow

# Parametricity

- Generic functions must behave uniformly for all possible type arguments
  - no `isInstanceOf` (it's broken anyway because of erasure)
- This means more restrictions
- But you can learn *much* more about a function from its signature
- This is called parametric reasoning/free theorems

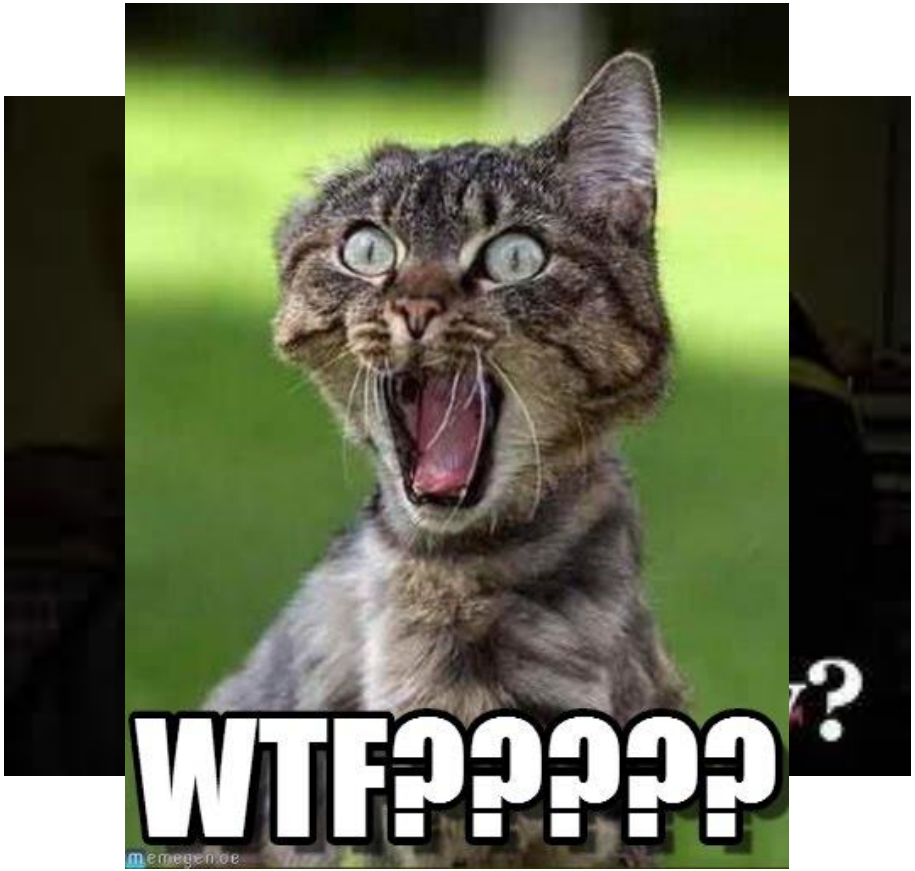
# Parametricity, contd.

- `def f[A]`
- What does
  - Is it refe
  - If so, the
- `def f[A]`
  - function
  - can't ad
  - can only
  - will do th sts on the input)
- learned all

# I/O in FP

# Now what?

- Totality
  - so I can’t loop forever?
  - no exceptions, so no errors?!
- Referential transparency
  - so I can’t use println?
- Determinism
  - so I can’t read input?
- How to get anything done?!



# So how do we do anything useful?

- Pure FP as defined before is useless for doing stuff
- But it’s really good at manipulating values
- So instead of writing a program that does stuff, create a value that contains instructions of what needs to be done
- some kind of run-time takes care of executing that description

We need:

- A type for these instructions: `IO`
- Some primitives to create such instructions, e. g. `putStrLn`
- Ways to combine these primitives into larger, more complex descriptions

# IO Monoid

- `val unit: IO`
  - instructions to do nothing
- `def putStrLn(s: String): IO`
  - doesn't print anything
  - returns instructions on how to print the String s
- `|+|` binary operator
  - takes two `IO` values and creates a new one
  - returned value is an instruction to perform the two instructions after each other
- This is already useful! e. g. 99 bottles of beer

```
val bottles: IO = (1 to 99).reverse.map { n =>
  putStrLn(s"$n bottles standing on a wall etc. pp.")
}.foldLeft(unit)(_ |+| _)
```

# `IO` Monad

- `IO` Monoid isn't powerful enough
  - output seems to work fine
  - no way to do input
- Need a way for later `IO` instructions to depend on the results of earlier instructions

# `IO` Monad, contd.

- Add a type parameter: `IO[A]`
  - a set of instructions that, when executed, will yield a value of type `A`
  - e. g. `readLine: IO[String]`
  - e. g. `putStrLn(s: String): IO[Unit]`
- And a new combinator `flatMap`
  - `flatMap: (IO[A], A => IO[B]) => IO[B]`
  - returns instructions to:
    - execute the `IO[A]`
    - feed the resulting value to the function
    - then execute the instructions returned by the function
- Also need `pure[A](a: A): IO[A]`
  - instructions to do nothing, just return the provided value

# IO Monad, contd.

- Now we have interactivity!

```
val helloWorld: IO[Unit] =
  putStrLn("Hi, what's your name?").flatMap { _ =>
    getStrLn.flatMap { name =>
      putStrLn(s"Hello, $name!")
    }
  }
```

- Slightly ugly due to nesting, but Scala has syntax sugar for this!

```
val helloWorld: IO[Unit] = for {
  _    <- putStrLn("Hi, what's your name?")
  name <- getStrLn
  _    <- putStrLn(s"Hello, $name")
} yield ()
```

- Again: executing this code will *not* cause anything to happen

# Guessing game

- That’s nice and everything, but how can we loop? Don’t we need state for that?
  - No, use recursion

```
def guessingGame(i: Int): IO[Unit] =
  getStrLn.flatMap { str =>
    val n = str.toInt // ask Hendrik about error handling ;-)
    if (n == i) {
       putStrLn("You win!")
    } else if (n < i) {
       putStrLn("Too low!") >> guessingGame(i)
    } else
       putStrLn("Too high!") >> guessingGame(i)
  }
```

`a >> b` is a shortcut for `a.flatMap(_ => b)`

# Effect primitives

- so far treated `putStrLn` and `getStrLn` as “magic”
- need a way to call side-effecting functions in a referentially transparent manner to e. g. use impure libraries
- specifics depend on the `IO` implementation
- several exist: scalaz-zio, cats-effect
- in cats-effect:

```
def putStrLn(s: String): IO[Unit] =
  IO.delay(println(s))
```

- other ways exist for different use cases (e. g. asynchronous IO)

# The end of the world

- so far we've only assembled instructions
- need to actually run them
- If you control the main function:

```
object MyApp extends IOApp {
  def run(args: List[String]): IO[ExitCode] =
    putStrLn("Hello, World!").as(ExitCode.Success)
}
```

- If you don't:

```
val myIO = putStrLn("Hello, World!")
myIO.unsafeRunSync()
```

# Why is this awesome?

- Concurrency
  - uniform treatment of synchronous and asynchronous I/O
  - very simple parallelism (e. g. `parMapN`)
  - light-weight threads (“Fibers”)
  - can safely interrupt threads without leaking resources
  - *way* more efficient than e. g. `scala.concurrent.Future`
  - don’t need an implicit `ExecutionContext`
  - built-in utilities for concurrent programming, e. g. queues, publish-subscribe, MVar, …
- I/O objects are first-class
  - can apply equational reasoning
  - transform them in arbitrary ways, e. g. write your own control structures!
  - many useful helper functions in libraries

# Error handling in FP

# Error handling in Functional Programming

Throwing exceptions is bad because

- Multiple exit points of a function
- Possible return values of a function are not seen in its signature
- Uncaught exceptions crash the program and there is no way for the compiler to warn you about that

# Alternative to throwing Exceptions

Dedicated data types

- **`Either[A,B]`**
  - Fail fast
  - `Right("computed value")` or `Left("failure description")`
- **`Validated[A,B]`**
  - error accumulation
  - `Valid("computed value")` or `Invalid(NonEmptyList("failure1", "failure2"))`

# Example: Get 42 divided by the min digit of a String

```
def minDigit(s: String): Int = s.sliding(1).map(_.toInt).min

def divide42By(i: Int): Int = 42 / i

def get42DividedByMinDigit: String => Int =
 minDigit _ andThen divide42By

try {
 get42DividedByMinDigit("51414")
} catch {
 case x: NumberFormatException => s"Number format exception ${x.getMessage}"
 case x: ArithmeticException   => s"Arithmetic exception ${x.getMessage}"
 case NonFatal(x)              => s"Unknown exception ${x.getMessage}"
}
```

Link

# With Either, we know that functions can fail

```
import cats.implicits._

def minDigit(s: String): Either[Throwable, Int] =
Either.catchNonFatal(s.sliding(1).map(_.toInt).min)

def divide42By(i: Int): Either[Throwable, Int] =
Either.catchNonFatal(42 / i)

def get42DividedByMinDigit(s: String): Either[Throwable, Int] =
 for {
   md <- minDigit(s)
   res <- divide42By(md)
 } yield res
```

# Unknown exception case still not nice

```
get42DividedByMinDigit(null) match {
 case Left(x: NumberFormatException) =>
    s"Number format exception ${x.getMessage}"
 case Left(x: ArithmeticException)   =>
    s"Arithmetic exception ${x.getMessage}"
 case Left(x)                        =>
    s"Unknown exception ${x.getMessage}"
 case Right(x)                       =>
    x.toString
}
```

Link

# Error model that allows to limit the possible errors

```
sealed trait AppError

case object EmptyString extends AppError

case object NonDigitsInString extends AppError

case object DivisionByZero extends AppError
```

# Function returns `Either[AppError, Int]`

```
import cats.implicits._
def minDigitSafe(s: String): Either[AppError, Int] = s match {
 case null | ""                  => Left(EmptyString)
 case _ if s.exists(!_.isDigit) => Left(NonDigitsInString)
 case _                          => Right(s.sliding(1).map(_.toInt).min)
}

def divide42By(i: Int): Either[AppError, Int] =
 Either.catchOnly[ArithmeticException](42 / i).leftMap(_ => DivisionByZero)

def get42DividedByMinDigit(s: String): Either[AppError, Int] =
 for {
   md <- minDigitSafe(s)
   i <- divide42By(md)
 } yield i
```

# Compiler checks if all error cases are handled

```
get42DividedByMinDigit("203") match {
 case Right(i)                => s"Min digit is $i"
 case Left(EmptyString)       => s"Empty string"
 case Left(DivisionByZero)    =>  s"Division by zero"
 case Left(NonDigitsInString) => s"Non digits in string"
}
```

Link

# How to combine IO with Either

- Throwables in IO
- MonadTransformers (EitherT)
- BifunctorIO

# Resources in FP

# Problem

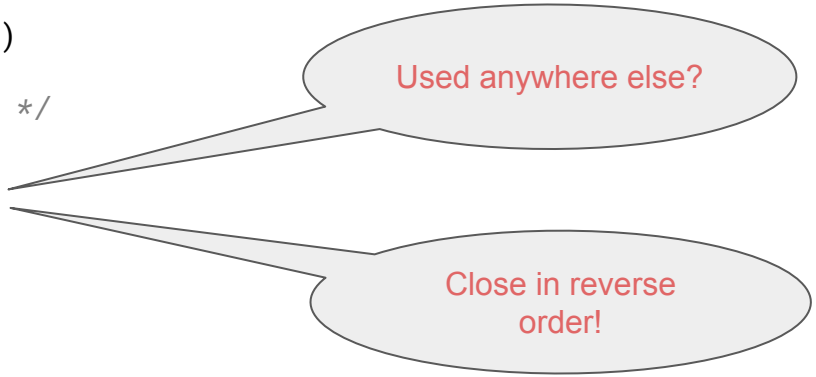
- Closing allocated resources is easily forgotten
  - especially in case of an error or
  - with nested resources
- `try-catch-finally` has its own problems
- Often, allocation and clean-up code end up in different places

# Example

```
def openStream(filename: String): Option[FileInputStream] =
 try {
   Some(new FileInputStream(filename))
 } catch {
   case _: FileNotFoundException => None
 }

def openReader(is: InputStream): Option[InputStreamReader] =
 try {
   Some(new InputStreamReader(is))
 } catch {
   case NonFatal(_) => None
 }

def doSomething(reader: InputStreamReader): Result = ???
```

# Example (contd.)

```
/* allocate resources */
val resources: Option[(InputStream, Option[InputStreamReader])] =
 openStream("foo").map(is => (is, openReader(is)))

try {
 resources.map(_._2.map(doSomething))
} catch {
 case NonFatal(_) => /* handle error */
} finally {
 resources.map(_._2.map(_.close))
 resources.map(_._1.close)
}
```



# A functional solution

- Resources need to be closed in any case
- Resources should be composable
- No `try-catch-finally`; no `throw`
- Resources should be usable, modifiable w/o losing the above properties

# cats.effect.Resource

- Create  a Resource:

```
def make[F[_], A](acquire: F[A])(release: A => F[Unit]): Resource[F, A] = ...
```

- Composition:

```
def flatMap[B](f: A => Resource[F, B]): Resource[F, B] = ...
```

- Use a Resource:

```
def use[B](f: A => F[B]): F[B] = ...
```

# Example revisited

```
def openStream(filename: String): Resource[IO, FileInputStream] =
 Resource.make { IO(new FileInputStream(filename)) } { r =>
   IO(r.close)
 }

def openReader(is: InputStream): Resource[IO, InputStreamReader] = ...

val r: Resource[IO, InputStreamReader] = for {
 is <- openStream("foo")
 reader <- openReader(is)
} yield reader

r.use(r => IO(doSomething(r)))

/* or even shorter: */
val p: IO[Result] = openStream("foo").use { ra =>
 openReader(ra).use { rb =>
   IO(doSomething(rb))
 }
}
```

# Summary

- Resources are guaranteed to be closed if:
  - program using the Resource is completed
  - acquisition of nested resources fails
  - acquisition of composed resources fails
  - using the resource produces an error
  - computation using the resource is cancelled (by another thread!)

# Questions?

Matthias.Berndt@cognotekt.com
Hendrik.Guhlich@cognotekt.com
Daniel.Beck@cognotekt.com