IO монада
ZIO
Cats Effect

Чистые функции

- Тотальные – для всех входных значений существует результат
- Детерминистичные – для одних и тех же входных данных один и тот же результат
- Отсутсвие сайд эффектов – функция не должна осуществлять операции ввода – вывода, изменять глобальные переменные, изменять входные параметры, т.е. не должна осуществлять взаимодействие с внешним миром

Плюсы чистых функций

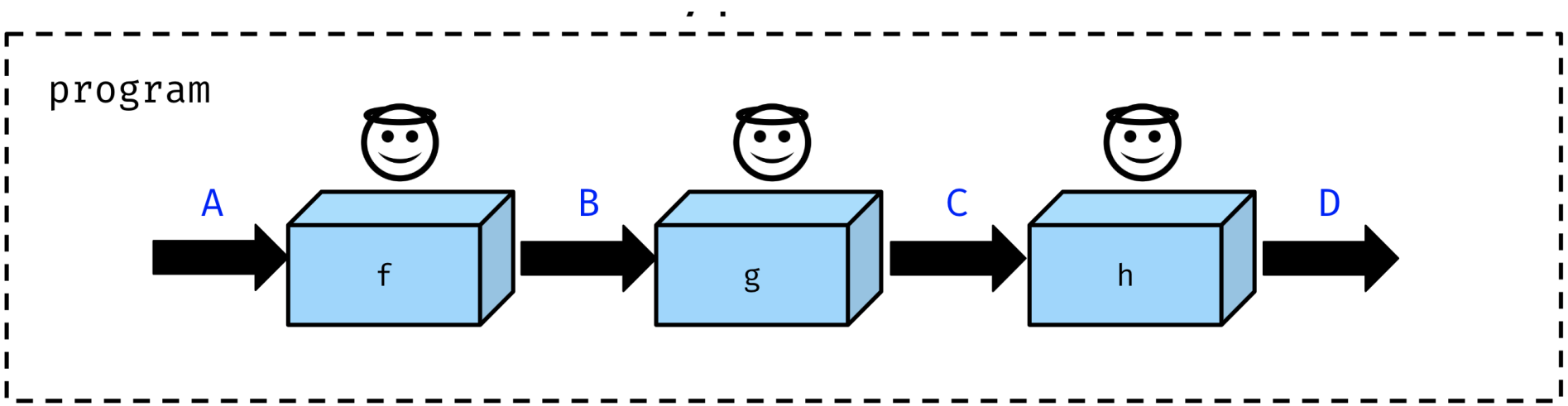
- Легко тестировать
- Сигнатура функции даёт более исчерпывающую информацию о функции
- Легко рефакторить
- Легко композировать

Ссылочная прозрачность

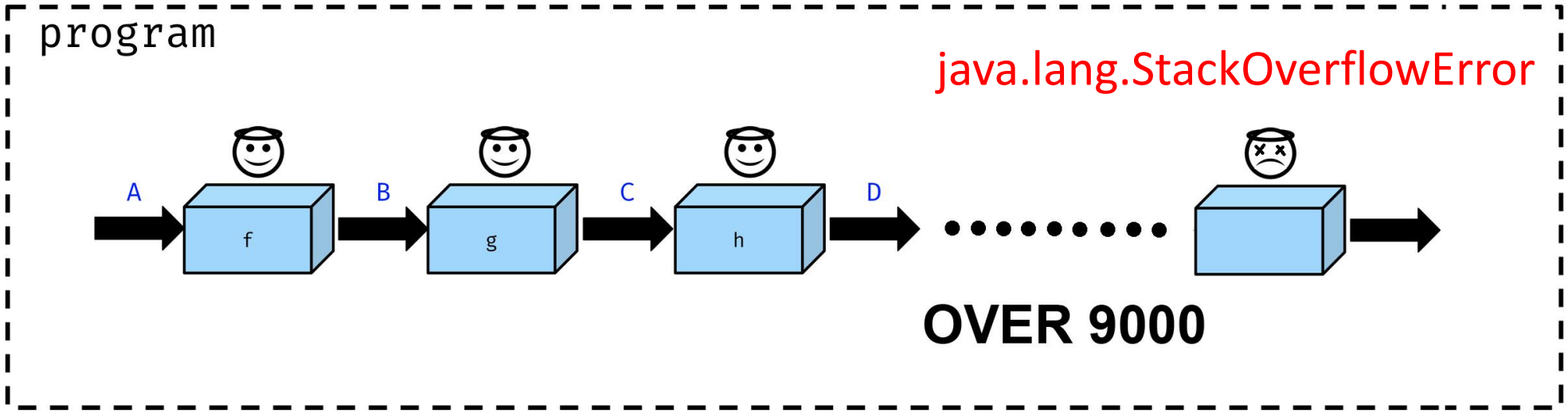
- Свойство выражения иметь возможность быть заменённым своим значением без изменения поведения программы

```
foo(42) + foo(42)

val x = foo(42)
x + x
```

Идеальный мир



Большое число вызовов (реальный мир)



Ограничения хвостовой рекурсии

- Хвостовая рекурсия работает только для само рекурсивных функций

```
def even(i: Int): Boolean = i match {
  case 0 => true
  case _ => odd(i - 1)
}

def odd(i: Int): Boolean = i match {
  case 0 => false
  case _ => even(i - 1)
}
```

- Чтобы её использовать нужно чтобы выражение, из которого вычисляется результат, если оно содержит рекурсивный вызов, состояло только из этого вызова

```
def unsafeFac(n: Int): Int =
  if (n == 0)  1
  else n * unsafeFac(n - 1)
```

Trampolining

- Основная идея – сделать, чтобы каждая функция (even, odd) возвращала continuation, который представляет следующий вызов или окончательный результат вычисления. Эти функции будут вычисляться в цикле, пока не будет получен результат
- Continuation представляет собой thunk – функцию без аргументов, содержащую оставшуюся часть вычислений

Simple trampoline

```scala
sealed trait Trampoline[+A] // ADT for holding either thunk or result

final case class Done[A](result: A) extends Trampoline[A]
final case class More[A](f: () => Trampoline[A]) extends Trampoline[A]

def run[A](t: Trampoline[A]): A = {

  var curr: Trampoline[A] = t
  var res: Option[A] = None

  while (res.isEmpty) {
    curr match {
      case Done(result) =>
        res = Some(result)
      case More(k) =>
        curr = k()
    }
  }
  res.get
}

def even(n: Int): Trampoline[Boolean] = {
  if (n == 0) Done(true)
  else More(() => odd(n - 1))
}

def odd(n: Int): Trampoline[Boolean] = {
  if (n == 0) Done(false)
  else More(() => even(n - 1))
}

println(run(even(100000001)))
```

```
curr == More(() => odd(100000001-1))

curr == More(() => even(100000000-1))
…
…
…
curr == More(() => odd(1-1))

curr == Done(false)
```

Tailrec loop

```scala
sealed trait Trampoline[+A]

final case class Done[A](result: A) extends Trampoline[A]
final case class More[A](f: () => Trampoline[A]) extends Trampoline[A]

@scala.annotation.tailrec
final def run[A](t: Trampoline[A]): A = {
  t match {
    case Done(result) => result
    case More(k) => run(k())
  }
}

def even(n: Int): Trampoline[Boolean] = {
  if (n == 0) Done(true)
  else More(() => odd(n - 1))
}

def odd(n: Int): Trampoline[Boolean] = {
  if (n == 0) Done(false)
  else More(() => even(n - 1))
}

println(run(even(100000001)))
```

Factorial

```
def fact(n: Int): Trampoline[Int] =
  if (n == 0) Done(1) else More(() => n * fact(n - 1))
```

- Для осуществления операций с полученным результатом одного из вызовов функции необходимо добавить case класс Cont и в цикле в функции run добавить структуру эмулирующую стек. Т.О. мы добиваемся эмуляции стека в heap

StackBase trampoline

```
final case class Cont[A, B](a: Trampoline[A], f: A => Trampoline[B]) extends Trampoline[B]

def run[A](t: Trampoline[A]): A = {
  var curr: Trampoline[Any] = t
  var res: Option[A] = None

  var stack: List[Any => Trampoline[A]] = List()
  while (res.isEmpty) {
    curr match {
      case Done(result) =>
        stack match {
          case Nil =>
            res = Some(result.asInstanceOf[A])
          case f :: rest =>
            stack = rest
            curr = f(result)
        }
      case More(k) =>
        curr = k()
      case Cont(a, f) =>
        curr = a
        stack = f.asInstanceOf[Any => Trampoline[A]] :: stack
    }
  }
  res.get
}

def fact(n: Int): Trampoline[Int] =
  if (n == 0) Done(1) else Cont[Int, Int](More(() => fact(n - 1)), res => Done(n * res))
```

```
run(fact(3))

curr == Cont(More(() => fact(3-1)), res => Done(3*res))
curr == More(() => fact(3-1)) stack == List(res => Done(3*res))

curr == Cont(More(() => fact(2-1)), res => Done(2*res))
curr == More(() => fact(1-1)) stack == List(res => Done(2*res),
                                            res => Done(3*res))

curr == Done(1) stack == List(res => Done(2*res),
                              res => Done(3*res))

curr == Done(2) stack == List(res => Done(3*res))

curr == Done(6)
```

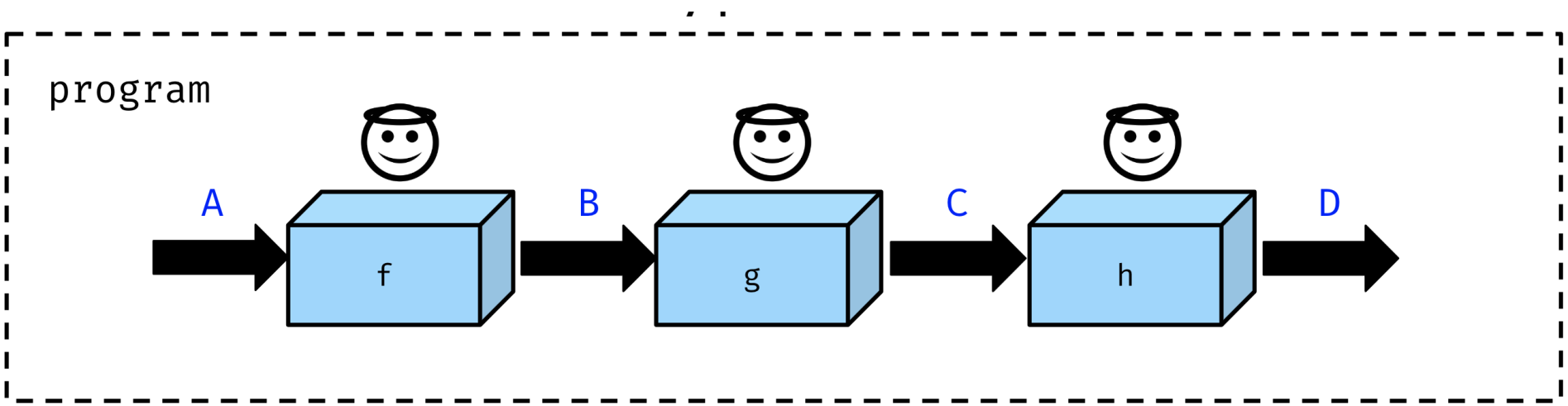
Tailrec loop

```scala
sealed trait Trampoline[+A]{
  def map[B](f: A => B): Trampoline[B] = flatMap(f andThen (Done(_)))
  def flatMap[B](f: A => Trampoline[B]): Trampoline[B] = Cont(this, f)
}

final case class Done[A](result: A) extends Trampoline[A]
final case class More[A](f: () => Trampoline[A]) extends Trampoline[A]
final case class Cont[A, B](a: Trampoline[A], f: A => Trampoline[B]) extends Trampoline[B]

@scala.annotation.tailrec
def run[A](tr: Trampoline[A]): A = tr match {
  case Done(a) => a
  case More(r) => run(r())
  case Cont(x, f) => x match {
    case Done(a) => run(f(a))
    case More(r) => run(Cont(r(), f))
    case Cont(y, g) => run(y.flatMap(g(_) flatMap f))
  }
}


def fact(n: Int): Trampoline[Int] =
  if (n == 0) Done(1)
  else More(() => fact(n - 1)).flatMap(res => Done(n * res))
```

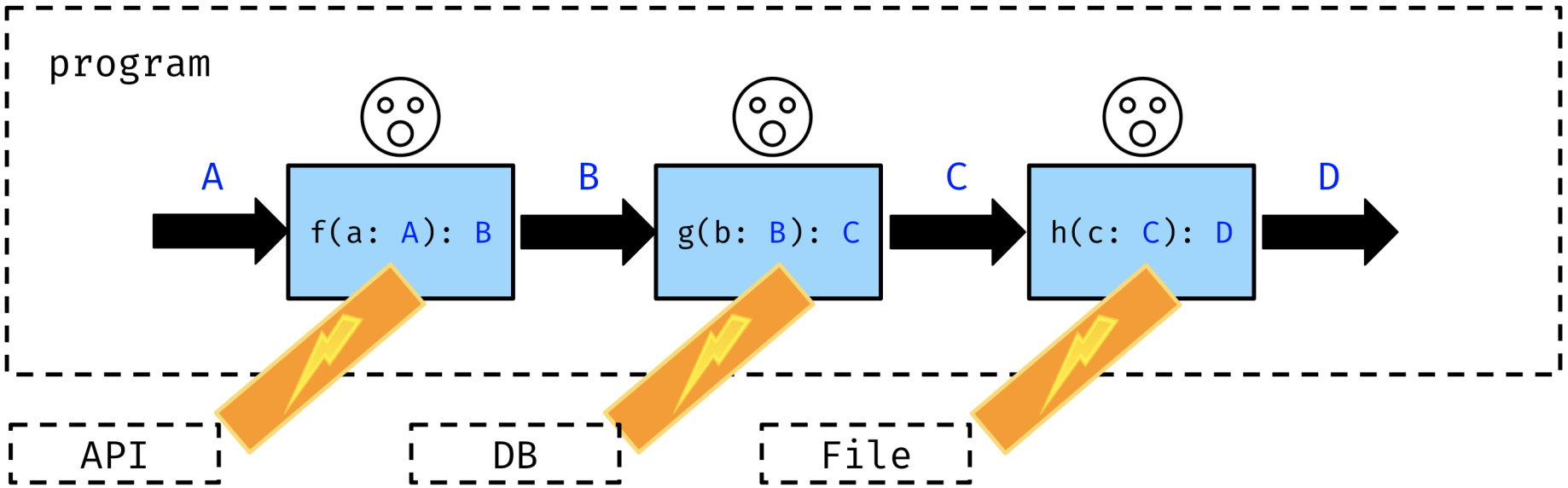
Scala.util.control.TailCalls

```scala
import scala.util.control.TailCalls._

def fac(n: Int): TailRec[Int] =
  if (n == 0)
    done(1)
  else
    for {
      x <- tailcall(fac(n - 1))
    } yield (n * x)
```

Идеальный мир

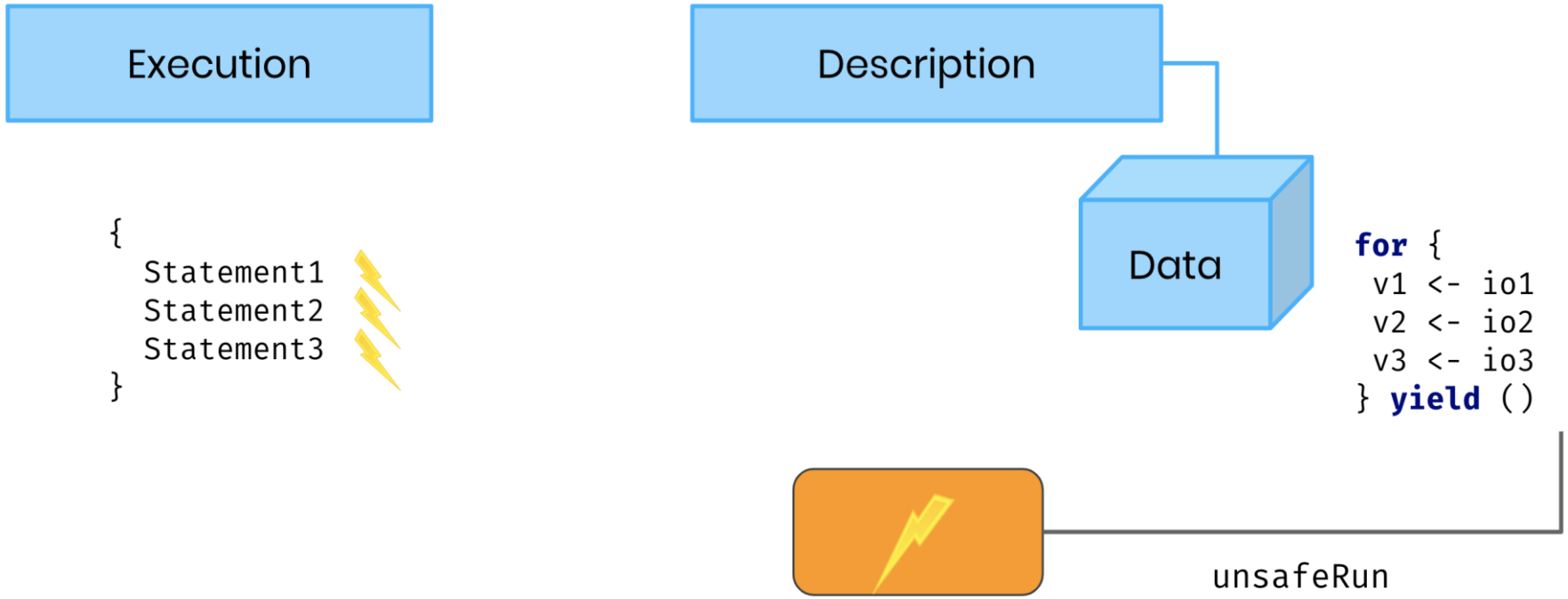


Сайд эффекты (реальный мир)



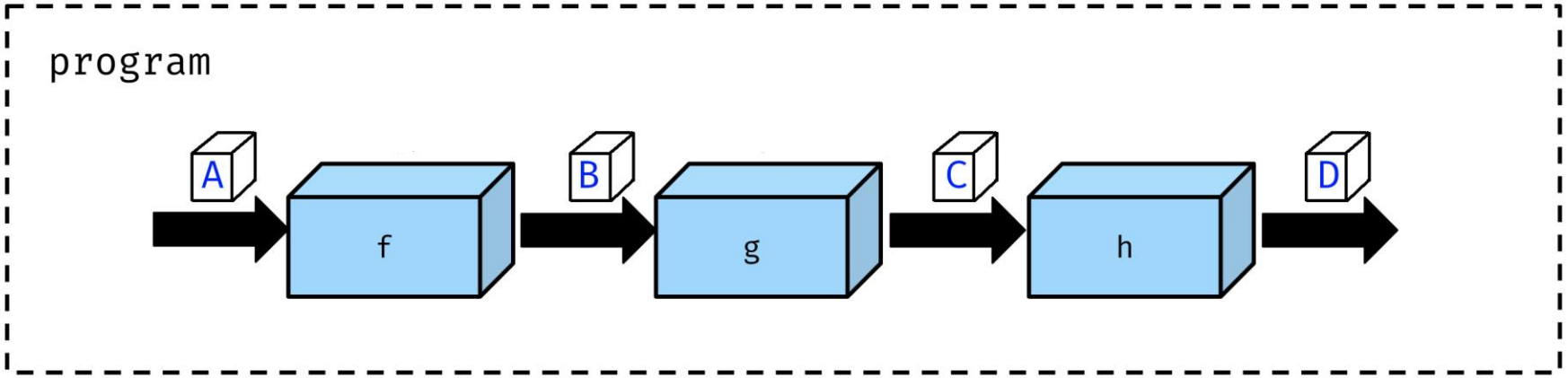
- Усложняют тестирование
- Усложняют композицию
- Усложняют организацию параллельной обработки данных
- По сигнатуре сложнее понять: что делает функция

Functional Effects (John A. De Goes)



- Функциональные эффекты – описание в виде иммутабельной структуры данных сайд эффектов.
- Вместо непосредственного осуществления сайд эффектов, программа строится на основе композиции функциональных эффектов. После чего в main (at the end of the world) осуществляется интерпретация функциональных эффектов, т.е. их преобразование в сайд эффекты (непосредственное взаимодействие с внешним миром) на основе их описания

Functional Effects



IO монада

- Позволяет строить чистые функции
- Обеспечивает синхронный FFI
- Обеспечивает последовательную композицию
- Поддерживает обработку ошибок
- Поддерживает асинхронность
- Поддерживает Concurrency
- Stack safety
- Resource safety

IO monads

- Cats Effect
- ZIO
- Monix

Haskell

```haskell
main :: IO ()
main = putStrLn "Hello, World!"
```

- FFI обёрнут в IO
- IO/runtime управляет concurrency
- Lazy evaluation
- Прототип main включает IO по умолчанию
- Tail call elimination

Scala

- Eager evaluation
- Никакого IO FFI к Java
- Main возвращает Unit
- Низкоуровневые Concurrency примитивы
- Ограниченные возможности по tail call elimination

IO monad API

- FFI – оборачивание сайд эффектов
- Комбинаторы – построение сложных IO посредством композиции более простых
- Runners – преобразование IO в сайд эффекты

Простейшее IO

```
// FFI
def delay[A](a: => A): IO[A]
// combinators
def pure[A](a: A): IO[A]
def flatMap[A, B](fa: IO[A])(f: A => IO[B]): IO[B]
// runners
def unsafeRunSync[A](fa: IO[A]): A
```

- Изоморфно () => A
- Data Type + Interpreter

IO as Data Type

```
sealed trait IO[+A]
case class FlatMap[B, +A](io: IO[B], k: B => IO[A]) extends IO[A]
case class Pure[+A](v: A) extends IO[A]
case class Delay[+A](eff: () => A) extends IO[A]
```

```
println("name?")
val n = readLine()
println(s"Hello $n")
```

```
FlatMap[String, Unit](
  FlatMap[Unit, String](
    Delay(() => println("name?")),
    _ => Delay[String](() => readLine())
  ),
  n => Delay[Unit](() => println(s"Hello $n"))
)
```

Runloop

```
def unsafeRunSync[A](io: IO[A]): A = {

  def loop(current: IO[Any], stack: Stack[Any => IO[Any]]): A =
    current match {
      case FlatMap(io, k) =>
        loop(io, stack.push(k))
      case Delay(body) =>
        val res = body() // запуск сайд эффектов
        loop(Pure(res), stack)
      case Pure(v) =>
        stack.pop match {
          case None => v.asInstanceOf[A]
          case Some((bind, stack)) =>
            val nextIO = bind(v)
            loop(nextIO, stack)
        }
    }
  loop(io, Stack.empty)
}
```

IO[A]

- Тип данных для представления сайд эффектов.
- Способен выразить как синхронные так и асинхронные вычисления
- Представляет вычисление, которое производит одно значение типа A, оканчивается неудачей или никогда не оканчивается
- Ссылочно прозрачен
- Immutable
- Множество алгебр (Monad, Concurrent…)

Делаем синхронный код ленивым и ссылочно прозрачным

```scala
def main(args: Array[String]): Unit = {
  import cats.effect.IO

  val ioa = IO { println("hey!") }
  val program: IO[Unit] =
    for {
      _ <- ioa
      _ <- ioa
    } yield ()

  program.unsafeRunSync()
  //=> hey!
  //=> hey!
}
```

Simple interactive app

```
def delay[A](body: => A): IO[A]
```

```
import cats.effect.IO
val program: IO[Unit] =
  for {
    _ <- IO.delay(println("name?"))
    n <- IO.delay(readLine())
    _ <- IO.delay(println(s"Hello $n"))
  } yield ()

program.unsafeRunSync()
```

Стекобезопасность и помещение значения в контекст

```
def pure[A](a: A): IO[A]
```

```
def fib(n: Int, a: Long = 0, b: Long = 1): IO[Long] =

  IO(a + b).flatMap { b2 =>
    if (n > 0)
      fib(n - 1, b, b2)
    else
      IO.pure(a)
  }
```

Описание асинхронного процесса

```
def async[A](k: (Either[Throwable, A] => Unit) => Unit)
```

```
def convert[A](fa: => Future[A])(implicit ec: ExecutionContext): IO[A] =
  IO.async { cb =>
    fa.onComplete {
      case Success(a) => cb(Right(a))
      case Failure(e) => cb(Left(e))
    }
  }
```

Отложенное выполнение IO.suspend

```
import cats.effect.IO

def fib(n: Int, a: Long, b: Long): IO[Long] =
  IO.suspend {
    if (n > 0)
      fib(n - 1, b, a + b)
    else
      IO.pure(a)
  }
```

Обработка ошибок

```scala
def unsafeRunSync[A](io: IO[A]): A = {
  def loop(current: IO[Any], stack: Stack[Bind]): A =
    current match {
      case FlatMap(io, k) =>
        loop(io, stack.push(Bind.K(k)))
      case HandleErrorWith(io, h) =>
        loop(io, stack.push(Bind.H(h)))
      case Delay(body) =>
        try {
          val res = body()
          loop(Pure(res), stack)
        } catch {
          case NonFatal(e) => loop(RaiseError(e), stack)
        }
      case Pure(v) =>
        stack.dropWhile(_.isHandler) match {
          case Nil => v.asInstanceOf[A]
          case Bind.K(f) :: stack => loop(f(v), stack)
        }
      case RaiseError(e) =>
        stack.dropWhile(!_.isHandler) match {
          case Nil => throw e
          case Bind.H(handle) :: stack => loop(handle(e), stack)
        }
    }
  loop(io, Nil)
}
```

```scala
case class RaiseError(e: Throwable) extends IO[Nothing]
case class HandleErrorWith[+A](io: IO[A], k: Throwable => IO[A])
  extends IO[A]
```

```scala
sealed trait Bind {
  def isHandler: Boolean = this.isInstanceOf[Bind.H]
}
object Bind {
  case class K(f: Any => IO[Any]) extends Bind
  case class H(f: Throwable => IO[Any]) extends Bind
}
```

Attempt

```
def attempt: IO[Either[Throwable, A]]
```

```
def getUserIdByEmail(string: String): IO[Long] =
  if (!string.contains("@"))
    IO.raiseError(new Exception("Invalid Email"))
  else
    IO.pure(1L)

def getUsersCosts(id: Long): IO[Array[Int]] =
  if (id == 1)
    IO.pure(Array[Int](1, 2, 3))
  else
    IO.raiseError(new Exception("There are no costs"))

def getReport(costs: Array[Int]): IO[String] =
    IO.pure("Mega report")
```

```
val email = "SomeEmail"
val program = for {
  id <- getUserIdByEmail(email)
  costs <- getUsersCosts(id)
  report <- getReport(costs)
} yield (report)

program.attempt.unsafeRunSync() match {
  case Right(report) => println(report)
  case Left(error) => println(s"Ops $error")
}
```

```
def getUserIdByEmail(string: String): Long =
  if (!string.contains("@"))
    throw new Exception("Invalid Email")
  else
    1L

def getUsersCosts(id: Long): Array[Int] =
  if (id == 1)
    Array[Int](1, 2, 3)
  else
    throw new Exception("There are no costs")

def getReport(costs: Array[Int]): String = "Mega report"
```

```
val email = "SomeEmail"
try{
  val id = getUserIdByEmail(email)
  val costs = getUsersCosts(id)
  val report = getReport(costs)
  println(report)
}
catch {
  case e:Throwable => println(s"Ops $e")
}
```

EitherT

```scala
def getUserIdByEmail(string: String): IO[Long] =
  if (!string.contains("@"))
    IO.raiseError(new Exception("Invalid Email"))
  else
    IO.pure(1L)

def getUsersCosts(id: Long): IO[Array[Int]] =
  if (id == 1)
    IO.pure(Array[Int](1, 2, 3))
  else
    IO.raiseError(new Exception("There are no costs"))

def getReport(costs: Array[Int]): IO[String] =
    IO.pure("Mega report")


val email = "SomeEmail"
val program = for {
  id <- getUserIdByEmail(email)
  costs <- getUsersCosts(id)
  report <- getReport(costs)
} yield (report)

program.attempt.unsafeRunSync() match {
  case Right(report) => println(report)
  case Left(error) => println(s"Ops $error")
}
```

```scala
sealed trait ReportError

case object InvalidEmail extends ReportError
case object ThereAreNoCosts extends ReportError

def getUserIdByEmail(string: String): IO[Either[ReportError, Long]] =
  if (!string.contains("@"))
    IO.pure(Left(InvalidEmail))
  else
    IO.pure(Right(1L))

def getUsersCosts(id: Long): IO[Either[ReportError,Array[Int]]] =
  if (id == 1)
    IO.pure(Right(Array[Int](1, 2, 3)))
  else
    IO.pure(Left(ThereAreNoCosts))

def getReport(costs: Array[Int]): IO[String] =
  IO.pure("Mega report")


val email = "SomeEmail"
val program = for {
  id <- EitherT( getUserIdByEmail(email))
  costs <- EitherT(getUsersCosts(id))
  report <- EitherT.liftF(getReport(costs))
} yield (report)

program.value.unsafeRunSync() match {
  case Right(report) => println(report)
  case Left(error) => println(s"Ops $error")
}
```

```scala
def main(args: Array[String]): Unit = {

  import scala.concurrent.Future
  import scala.concurrent.duration._
  import scala.concurrent.ExecutionContext.global
  implicit val contextGlobal: ExecutionContextExecutor = global

  val ioa = Future {
    println("hey!")
  }
  val program: Future[Unit] =
    for {
      _ <- ioa
      _ <- ioa
    } yield ()

  Await.result(program, 5.seconds)
  //=> hey!
}
```