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 должно осуществлять взаимодействие с внешним миром

Плюсы чистых функций

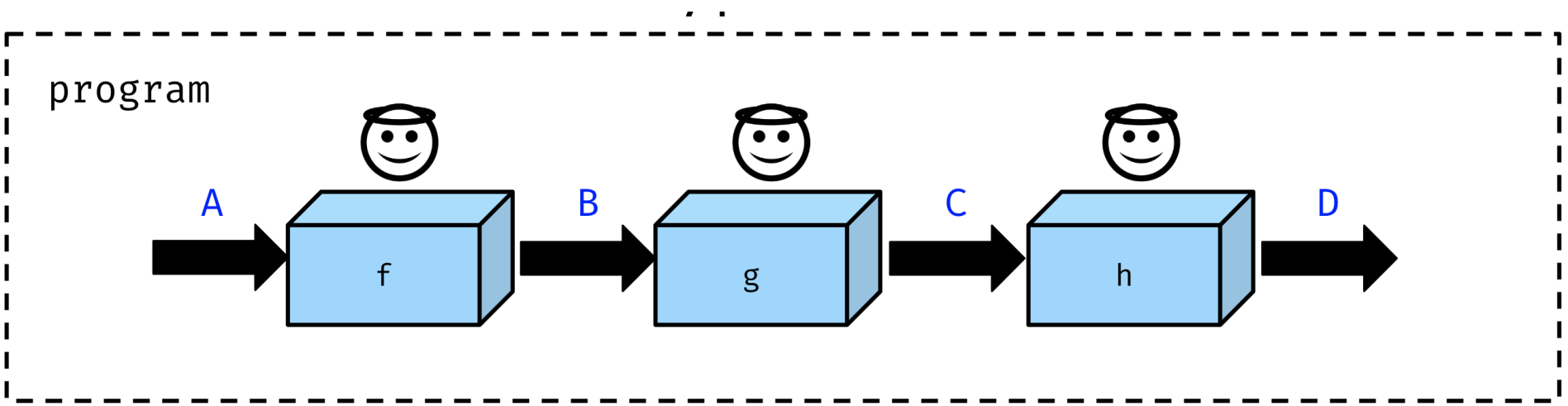
- Легко тестировать
- Сигнатура функции даёт более исчерпывающую информацию о функции
- Легко рефакторить
- Легко композировать

Ссылочная прозрачность

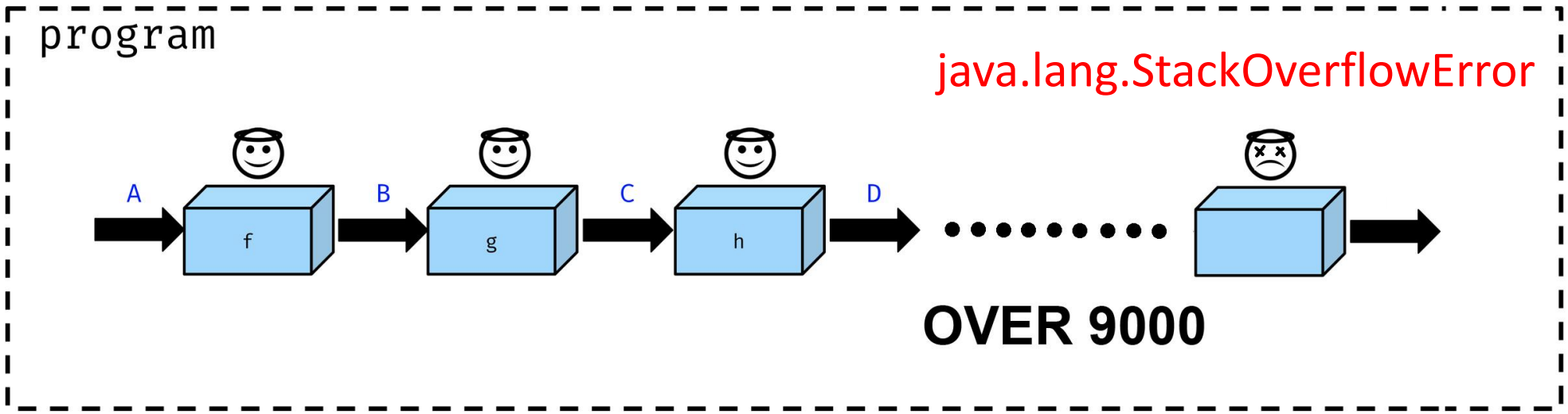
- Свойство выражения иметь возможность быть заменённым своим значением без изменения поведения программы

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

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

Идеальный мир



Большое число вызовов (реальный мир)



Ограничения хвостовой рекурсии

- Оптимизация хвостовой рекурсии 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

Trampoline data structure

```
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]
```

Simple trampoline

```
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)
```

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

Trampoline data structure

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

StackBase trampoline

```scala
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
}
```

```scala
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)
```

Trampoline data structure

```
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]
```

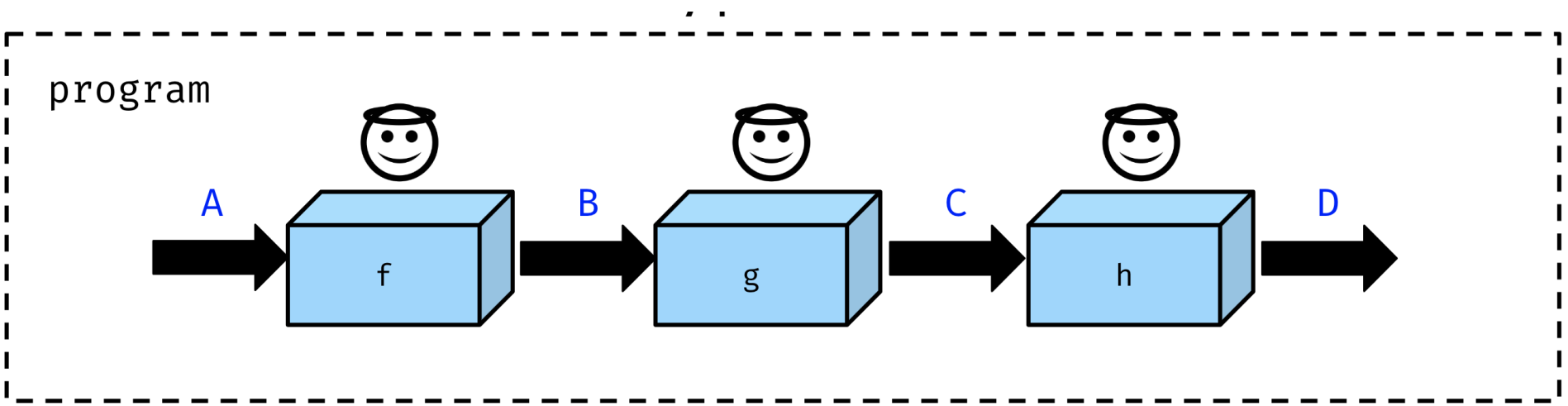
Tailrec run loop

```scala
@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))
```

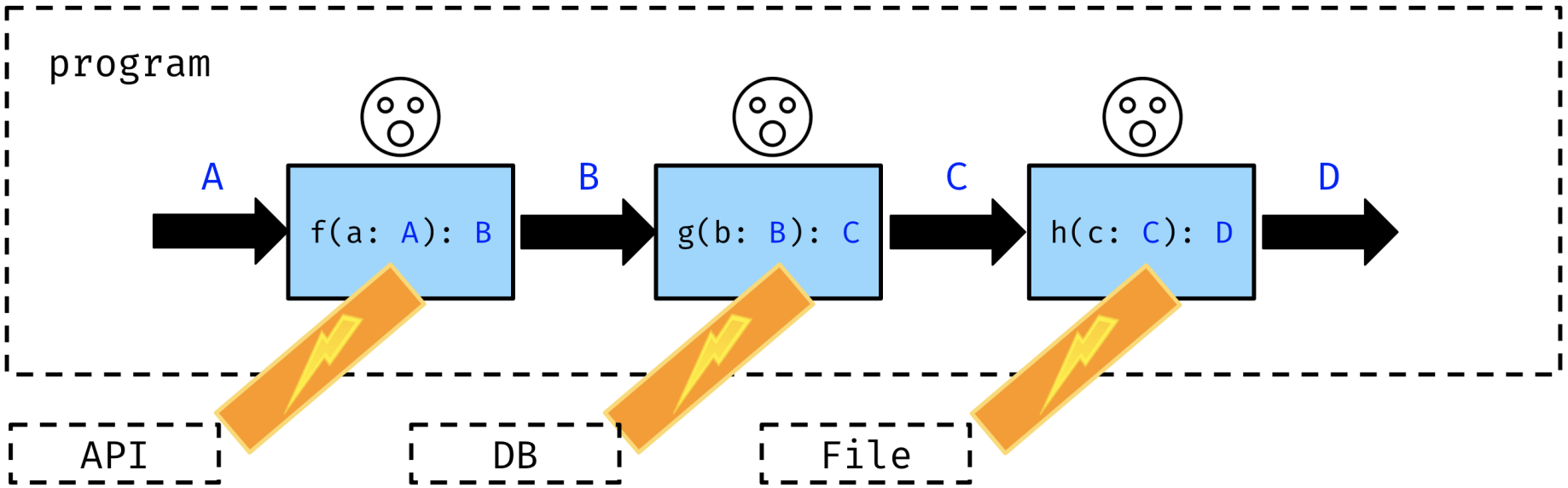
Scala.util.control.TailCalls

```
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

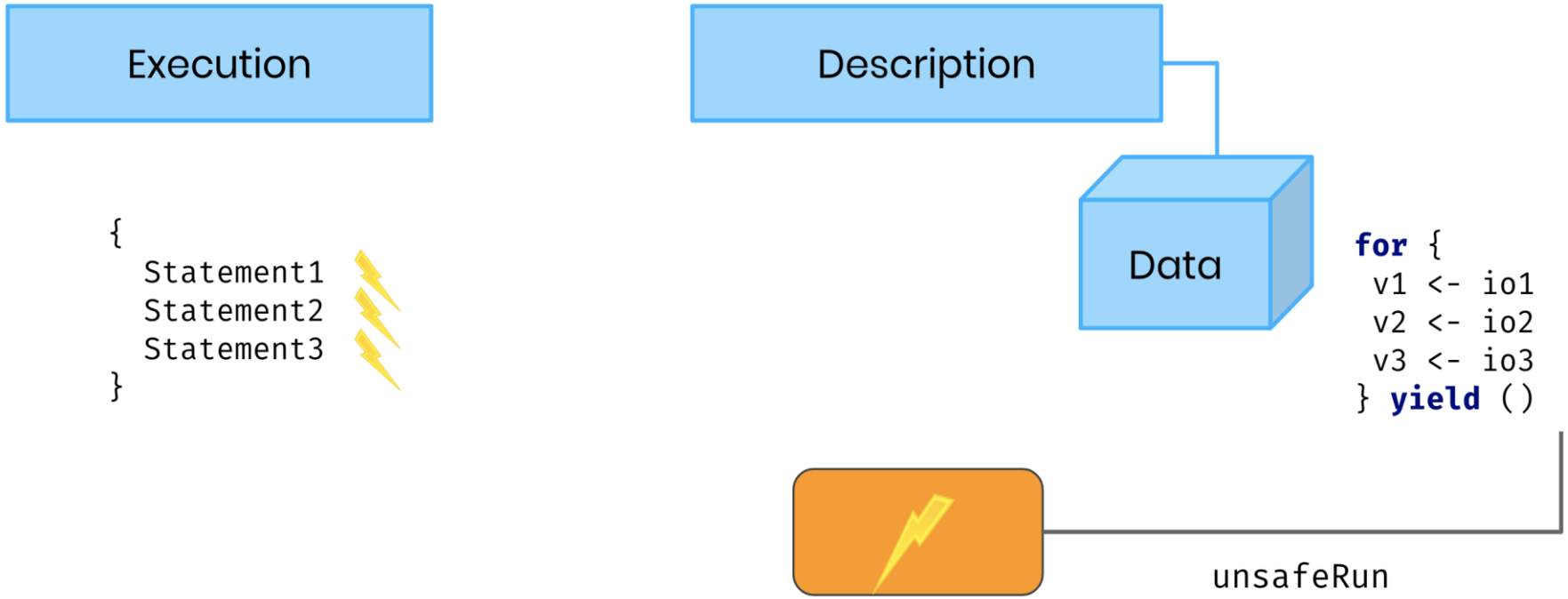


Сайд эффекты (реальный мир)



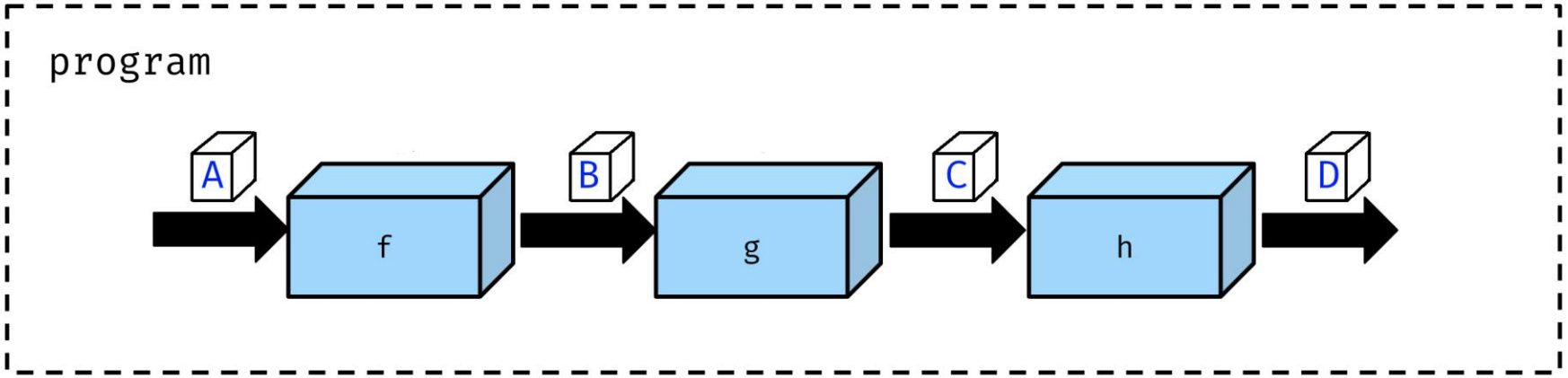
- Усложняют тестирование
- Усложняют композицию
- Усложняют организацию параллельной обработки данных
- По сигнатуре сложнее понять: что делает функция

Functional Effects (John A. De Goes)



- Функциональные эффекты – описание сайд эффектов в виде иммутабельной структуры данных.
- 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)

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

```scala
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]
```

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

```scala
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)
  }
```

IO as data type with error handling

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

```
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
}
```

Обработка ошибок

```
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)
}
```

Обработка ошибок в IO пример

```
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")
```

Attempt

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


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")
}
```

Аналогичная обработка ошибок в синхронном коде

```
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"
```

Аналогичная Обработка ошибок в синхронном коде. Сравнение

```
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")
}
```

```
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")
}
```

Custom Error type

```
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")
```

EitherT

```
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")
}
```

ZIO[R,E,A]

- R – тип окружения, которое требует эффект
- E – тип ошибки
- A – тип значения при успешном завершении успешного завершения

Значение типа ZIO[R,E,A] описывает вычисления вида R => Either[E, A]

Type Aliases

- UIO[A] – алиас для ZIO[Any, Nothing, A], который представляет эффект, который не требует никакого окружения, не может завершиться ошибкой и возвращает значение типа A
- URIO[R, A] – алиас для ZIO[R, Nothing, A]
- Task[A] – алиас для ZIO[Any, Throwable, A]
- RIO[R, A] – алиас для ZIO[R, Throwable, A]
- IO[E, A] – алиас для ZIO[Any, E, A]

Пример

```scala
import zio._

sealed trait ReportError

case object InvalidEmail extends ReportError
case object ThereAreNoCosts extends ReportError

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

def getUsersCosts(id: Long): IO[ReportError, Array[Int]] =
  if (id == 1)
    IO.succeed(Array[Int](1, 2, 3))
  else
    IO.fail(ThereAreNoCosts)

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

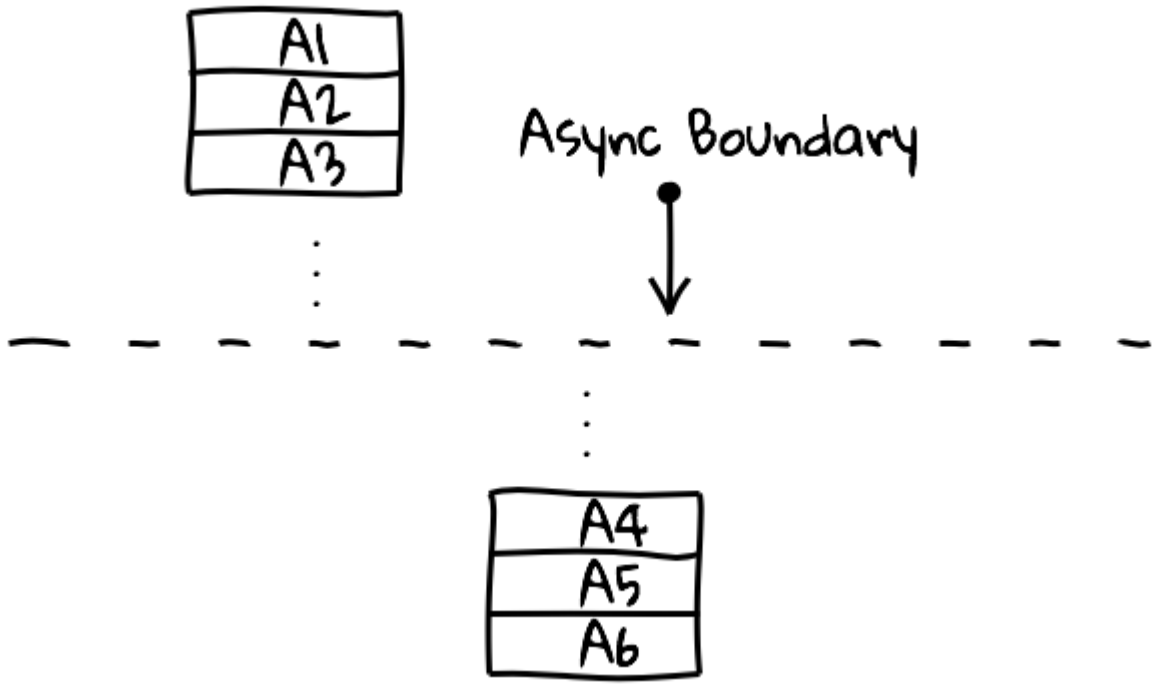
Пример

```
import zio._

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

val runtime = new DefaultRuntime {}

runtime.unsafeRunSync(program) match {
  case Exit.Failure(cause) => println(s"Ops $cause")
  case Exit.Success(value) => println(value)
}
```

Асинхронный процесс

- Процесс, продолжающий своё выполнение в другом месте или в другое время по отношению к тому где он стартовал

Асинхронный процесс

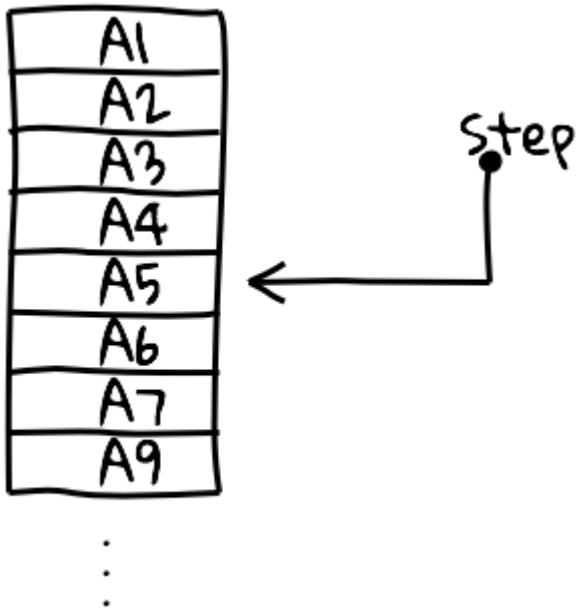


Concurrency

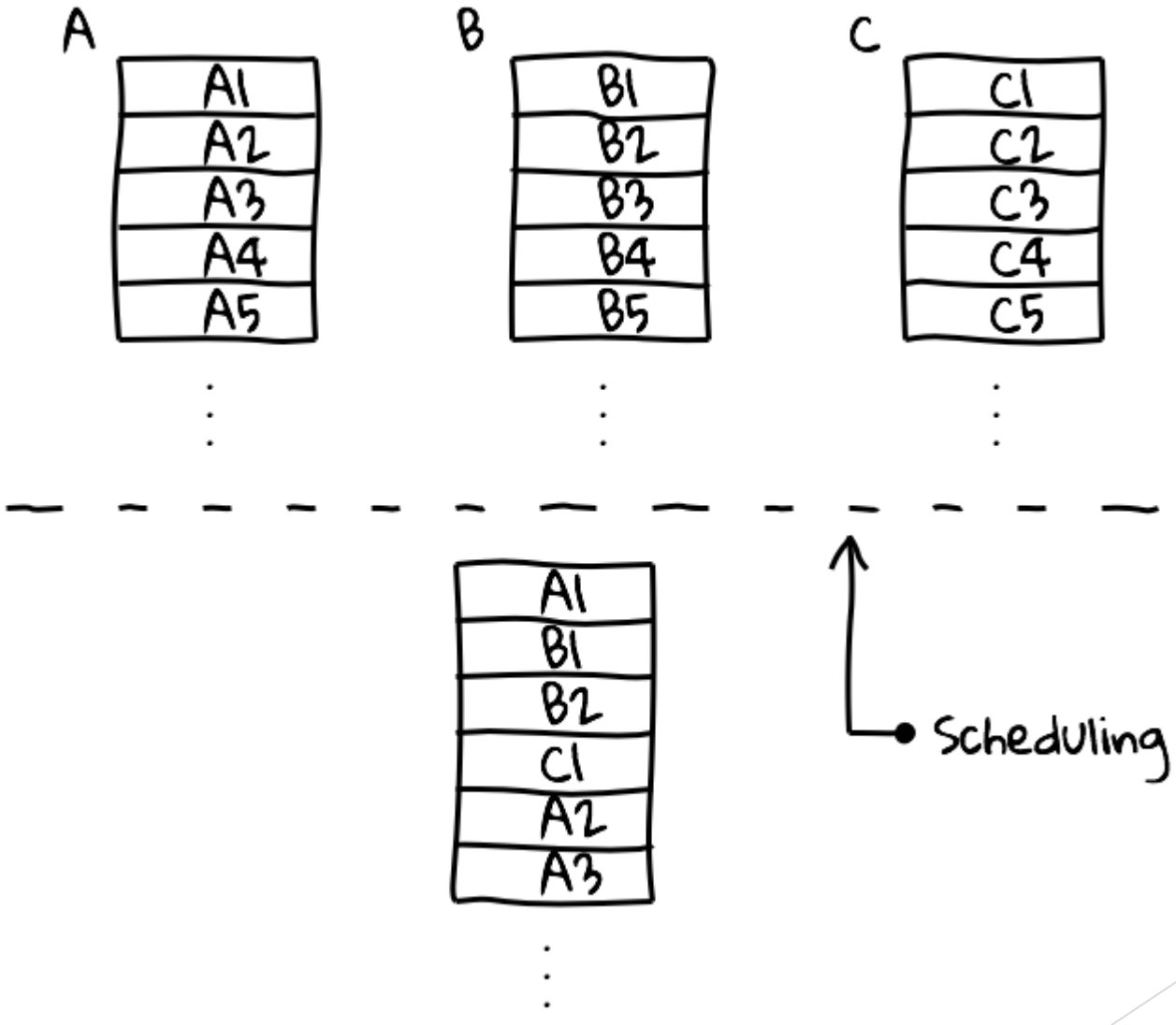
- Техника структурирования программы, в которой есть несколько логических потоков управления, чьи эффекты оказываются перемешанными

Логический поток

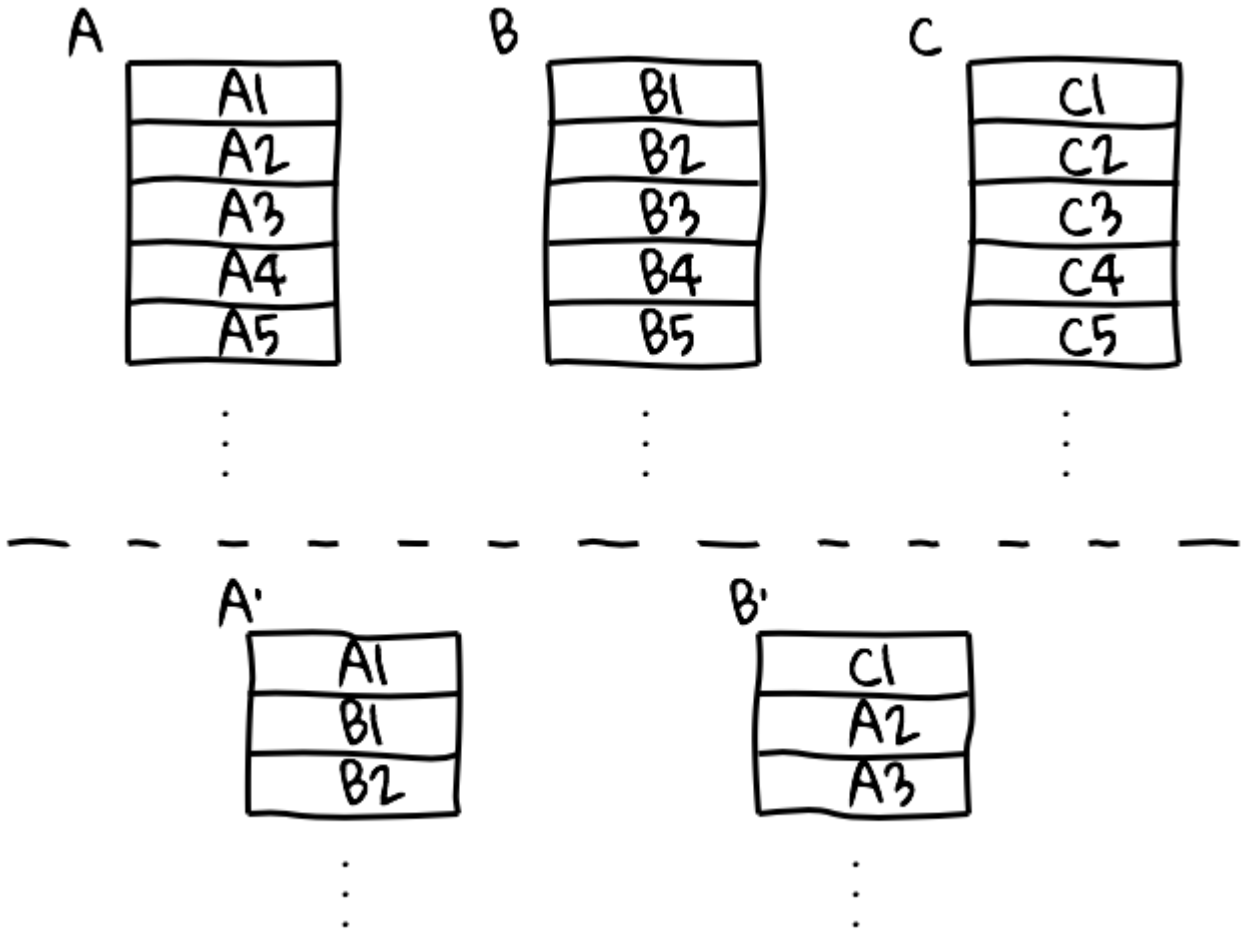
- Логический поток – последовательность дискретных шагов



Перемешивание

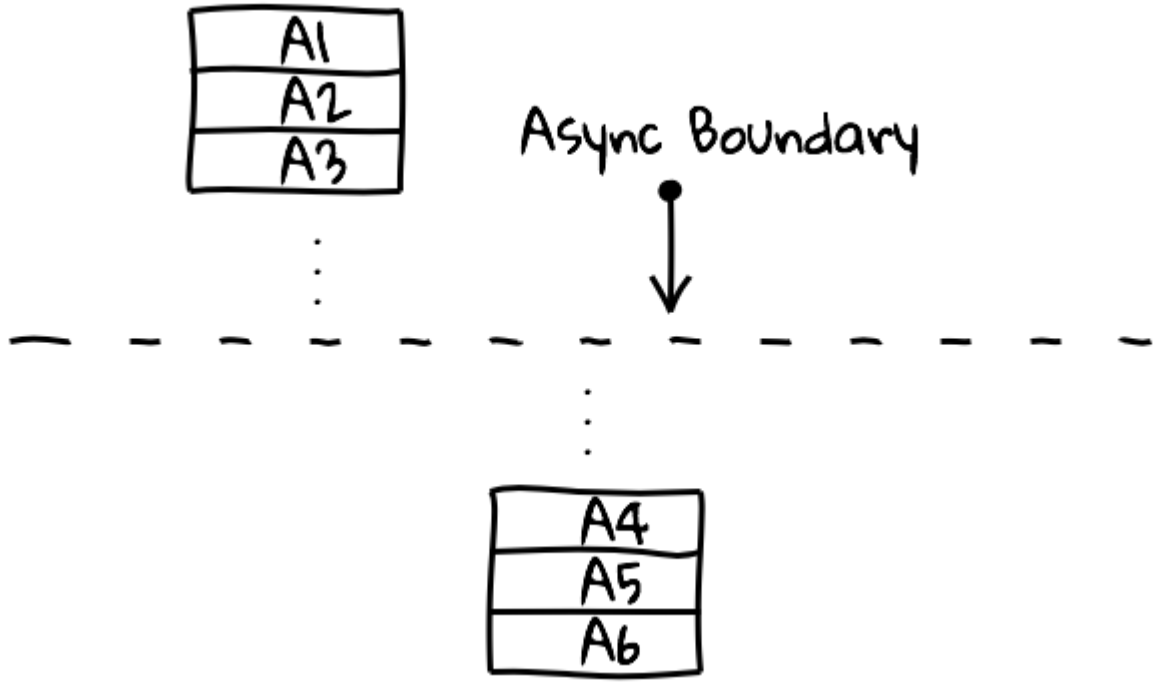


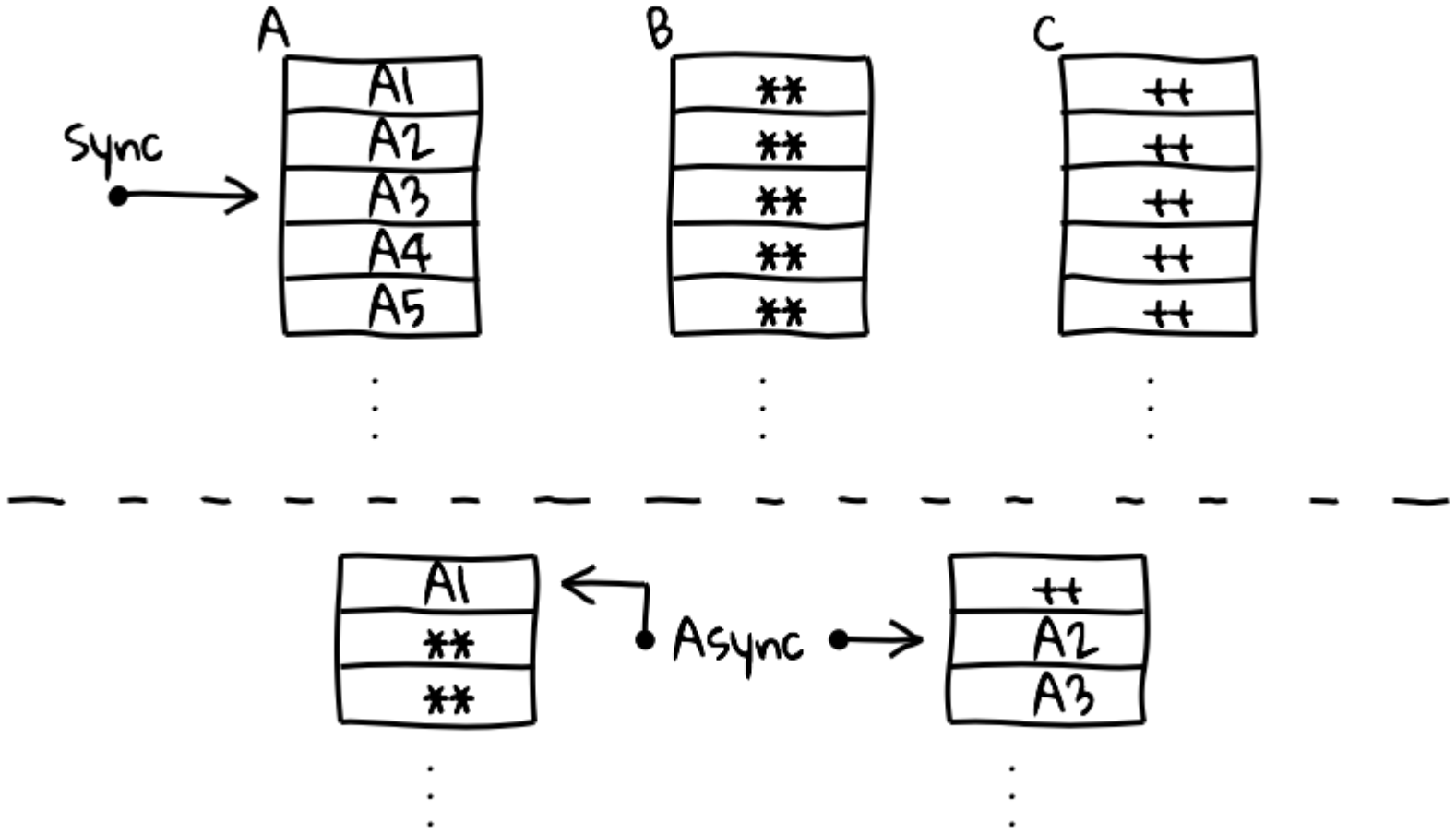
M:N Threading

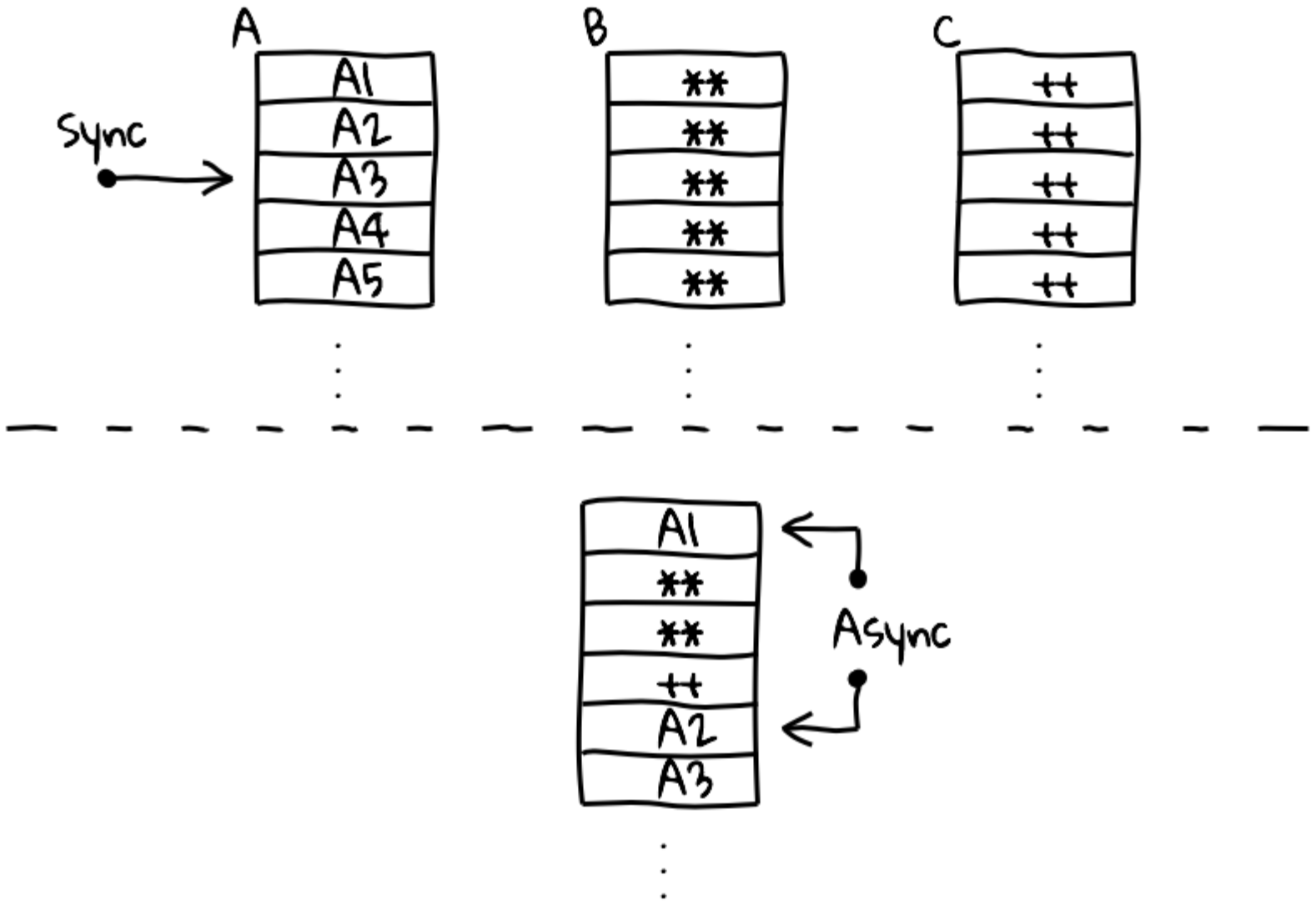


Потоки - абстракции

- Логический поток – предоставляет синхронный интерфейс к асинхронному процессу

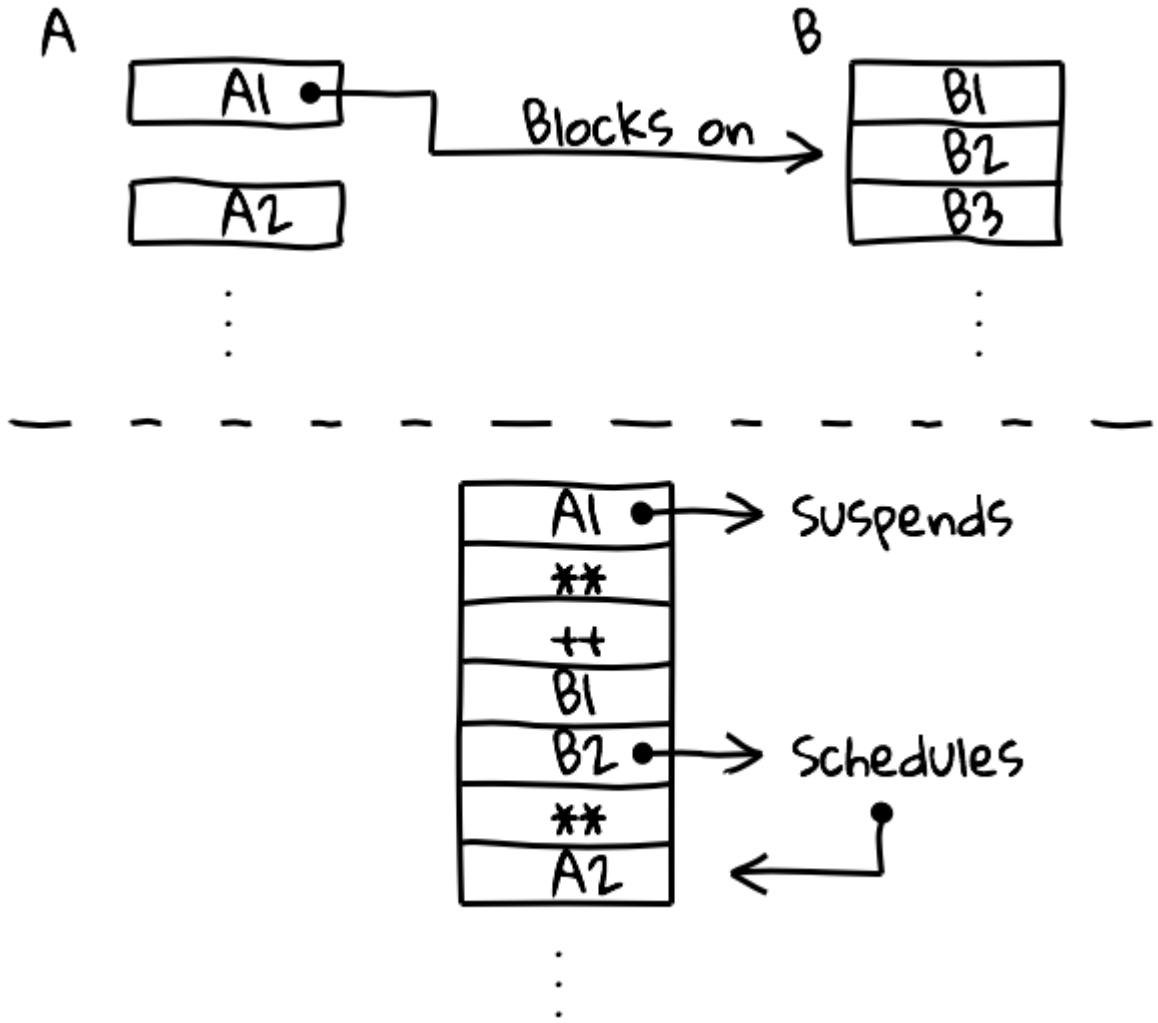

A1
A2
A3
Async Boundary
A4
A5
A6


A
B
C
Sync
A1
A2
A3
A4
A5
**
**
**
**
**
++
++
++
++
++
A1
**
**
Async
++
A2
A3


A
A1
A2
A3
A4
A5
Sync
B
**
**
**
**
**
C
++
++
++
++
++
A1
**
**
++
A2
A3
Async

Блокировка

- Блокировка на одном из уровней означает временное прекращение работы по данной задаче на более низком уровне (suspending), при этом потоки на более низком уровне продолжают работать



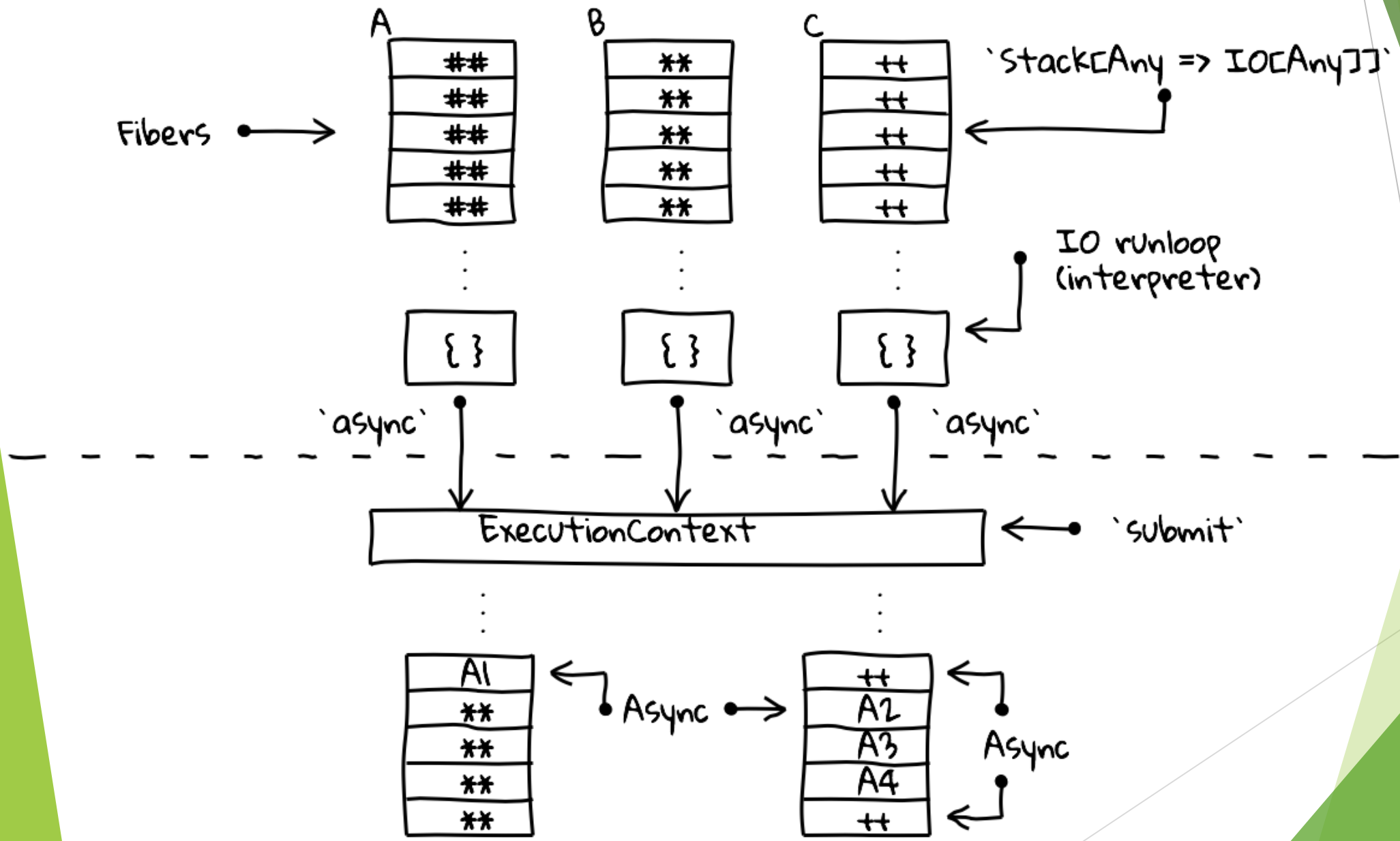
Уровни

- Процессы ОС – M:N с процессорами. Собственное состояние выполнения, собственное пространство памяти
- ОС/JVM Threads – M:N с процессами. Собственное состояние выполнения, разделяемое пространство памяти
- Fibers – M:N с потоками. Разделяемое состояние выполнение, разделяемое пространство памяти

Семантические блокировки

- JVM потоки – дефицитный ресурс
- Fibers – более дешёвый ресурс
- Блокировка на уровне Fiber не вызывает блокировку нижележащего потока JVM

Кооперативное планирование Fiber’ов



Fibers

```
def start[A](io: IO[A]): IO[Fiber[IO, A]]

trait Fiber[F[_], A] {
  def join: F[A]
  def cancel: F[Unit]
}


def sleep(duration: FiniteDuration)(implicit timer: Timer[IO]): IO[Unit]

def race[A, B](lh: IO[A], rh: IO[B])(implicit cs: ContextShift[IO]): IO[Either[A, B]]
```

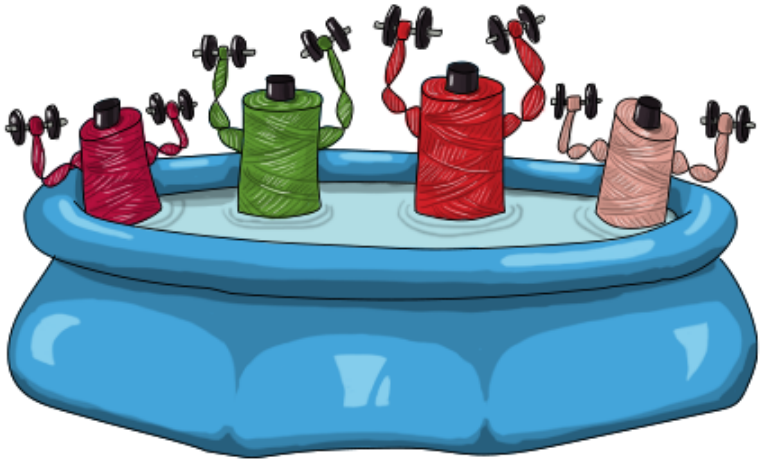
ContextShift

```scala
import scala.concurrent.ExecutionContext

trait ContextShift[F[_]] {

  def shift: F[Unit]

  def evalOn[A](ec: ExecutionContext)(f: F[A]): F[A]
}
```

- Аналог executionContext
- Предоставляет средства для кооперативной многозадачности
- Представляет средства для смены пула потоков для выполнения операций, например выполнение блокируемых операций (jdbc, file io)

THREAD POOL BEST PRACTICES

WORK-STEALING
CPU-BOUND (#CPUS)



COMPUTATION

FINITE RESOURCES
AVOID BLOCKING AT ALL COSTS

CACHING
UNBOUNDED SIZE

BLOCKING IO

HIGHEST PRIORITY
1 OR COUPLE OF THREADS

AVOID WORK AT ALL COSTS

EVENT DISPATCHER

DISCLAIMER:
TEST AND MEASURE!
WHEN IT COMES TO CONCURRENCY,
NOBODY HAS IDEA WHAT THEY'RE DOING.

Shift

- Эффект, который вызывает логический fork. Например мы хотим, чтобы долгие задачи не занимали поток на долгое время

```
import cats.effect._
import cats.implicits._

def fib(n: Int, a: Long = 0, b: Long = 1)
             (implicit cs: ContextShift[F]): IO[Long] = {

  IO.suspend {
    val next =
      if (n > 0) fib(n - 1, b, a + b)
      else IO.pure(a)

    // Triggering a logical fork every 100 iterations
    if (n % 100 == 0)
      cs.shift *> next
    else
      next
  }
}
```

Simple example

```
import cats.effect.{ContextShift, IO}
import scala.concurrent.ExecutionContext
implicit val contextShift: ContextShift[IO] =
   IO.contextShift(ExecutionContext.global)


val jobOne: IO[Int] = IO(???)
val jobTwo: IO[String] = IO(???)
for {
  j1Fiber <- jobOne.start
  j2Fiber <- jobTwo.start
  i <- j1Fiber.join
  s <- j2Fiber.join
} yield (i,s)
```

parMapN

```scala
import cats.effect.{ContextShift, IO}
import cats.implicits._

import scala.concurrent.ExecutionContext

implicit val contextShift: ContextShift[IO] =
IO.contextShift(ExecutionContext.global)

val ioA = IO(println("Running ioA"))
val ioB = IO(println("Running ioB"))
val ioC = IO(println("Running ioC"))

val program = (ioA, ioB, ioC).parMapN { (_, _, _) => () }

program.unsafeRunSync()
//=> Running ioB
//=> Running ioC
//=> Running ioA
```

parSequence

```scala
import cats.data.NonEmptyList
import cats.effect.{ContextShift, Timer, IO}
import cats.syntax.parallel._

import scala.concurrent.ExecutionContext

implicit val cs: ContextShift[IO] = IO.contextShift(ExecutionContext.global)

val anIO = IO(1)

val aLotOfIOs = NonEmptyList.of(anIO, anIO)

val ioOfList: IO[NonEmptyList[Int]] = aLotOfIOs.parSequence
```

parTraverse

```
import cats.data.NonEmptyList
import cats.effect.{ContextShift, Timer, IO}
import cats.syntax.parallel._

import scala.concurrent.ExecutionContext

implicit val cs: ContextShift[IO] = IO.contextShift(ExecutionContext.global)


val results:IO[NonEmptyList[Int]]  = NonEmptyList.of(1, 2, 3).parTraverse { i =>
  IO(i)
}
```

Безопасная работа с ресурсами

```
import java.io._
def javaReadFirstLine(file: File): String = {
  val in = new BufferedReader(new FileReader(file))
  try {
     in.readLine()
   } finally {
     in.close()
   }
}
```

- Сайд эффект, который сложно использовать для ФП
- Сложно использовать для асинхронных операций
- Если exception в finally то он перекрывает основной exception

IO.bracket

```
import java.io._
import cats.effect.IO
import cats.syntax.functor._
def readFirstLine(file: File): IO[String] =
  IO(new BufferedReader(new FileReader(file))).bracket { in =>
    // Usage (the try block)
     IO(in.readLine())
   } { in =>
     // Releasing the reader (the finally block)
     IO(in.close()).void
}
```

- Может использоваться в ФП
- Работает с асинхронными действиями
- Секция release выполнится независимо от результата use, будет ли это успешное завершение или ошибка или отмена выполнения
- Если use кидает ошибку и есть Ошибка внутри release будет, то ошибка внутри release выведена в std.err, а основной останется ошибка, которая была в use

Cancellation

- Не все IO отменяемые
- Статус отмены проверяется только после asynchronous boundary
- Достичь этого можно следующими способами
  - Построить IO с IO.cancelable, IO.async, IO.bracket
  - Использовать IO.cancelBoundary или IO.shift
- После asynchronous boundary проверка на cancelation идёт на каждом 512 flatMap

Cancelation должен поддерживаться

```
import cats.effect.{ContextShift, Timer, IO}
import scala.concurrent.ExecutionContext
implicit val cs: ContextShift[IO] = IO.contextShift(ExecutionContext.global)

val program = for {
  aFiber <- IO {
    Thread.sleep(1000)
    println("Still alive!!")
  }.start
  _ <- aFiber.cancel
} yield ()

program.unsafeRunSync()

//Still alive!!
```

cancelBoundary

```
import cats.implicits._
import cats.effect.{ContextShift, Timer, IO}
import scala.concurrent.ExecutionContext
implicit val cs: ContextShift[IO] = IO.contextShift(ExecutionContext.global)

val program = for {
  aFiber <- (IO {Thread.sleep(1000)} *>
             IO.cancelBoundary *>
             IO{println("Still alive!!")}).start
  _ <- aFiber.cancel
} yield ()

program.unsafeRunSync()
```

cancelBoundary

```scala
import cats.effect.IO
import cats.syntax.apply._

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

      // Every 100-th cycle check cancellation status
      if (n % 100 == 0)
        IO.cancelBoundary *> next
      else
        next
    }
  }
```

LIFT IO

+ CONVERT IO[A] INTO F[A]

EFFECT

+ LAZY AND ASYNC EVALUATION

BRACKET

+ SAFELY ACQUIRE AND RELEASE RESOURCES

ASYNC

+ ASYNCHRONOUS SUSPEND

CONCURRENT EFFECT

+ CANCELABLE AND CONCURRENT EVALUATION

SYNC

+ SYNCHRONOUS SUSPEND

CONCURRENT

+ CONCURRENTLY START OR CANCEL

CATS-EFFECT 1.0

Отсутствие ссылочной прозрачности для Future

```
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!
}
```

Последовательное и параллельное выполнение Future

```
// последовательное выполнение
def jobOne: Future[Int] = Future[Int](???)
def jobTwo: Future[String] = Future[String](???)

jobOne.flatMap(i=>jobTwo.map(s=>(i,s)))
```

```
// параллельное выполнение
val jobOne: Future[Int] = Future[Int](???)
val jobTwo: Future[String] = Future[String](???)

jobOne.flatMap(i=>jobTwo.map(s=>(i,s)))
```

Future vs IO

- IO – значение описывающие некоторое действие (возможно асинхронное)
- Future – дескриптор для доступа к результату уже запущенного действия (возможно асинхронного)

Future vs IO скорость

IO

- Thread shift по запросу
- Есть возможности делать thread shift для того чтобы потоки не занимались долгими операциями (fairness)
- По бенчмаркам более быстро работает

Future

- Thread shift на каждый map/flatMap
- Может быть сконфигурирована только через указание Execution context

Future vs IO cancellation

- Future не может быть отменена и остановлена после создания, что приводит к трате ресурсов
- IO может быть fork’нуто, после чего на нём можно вызвать join или cancel