Tagless Final

Олег Нижников

Tinkoff

План

1. ФП
2. Теоркат
3. Больше теорката
4. ФП

План

1. ФП
2. ~~Теоркат~~ Интуиция
3. ~~Больше теоркат~~ Интуиции
4. ФП

Функциональное программирование

Функции

```
val a: A
expr : B
```

Функциональное программирование

Функции

```
val a: A
expr : B
```

```
a => expr : A => B
```

Функциональное программирование

Функции

```
val a: A
expr : B
```

```
a => expr : A => B
```

```
def foo(a: A): B = expr
```

Тотальность

Функция возвращает результат для любого значения входного параметра

за конечное время

Тотальность

Функция возвращает результат для любого значения входного параметра

за конечное время

~~исключения~~

Тотальность

Функция возвращает результат для любого значения входного параметра

за конечное время

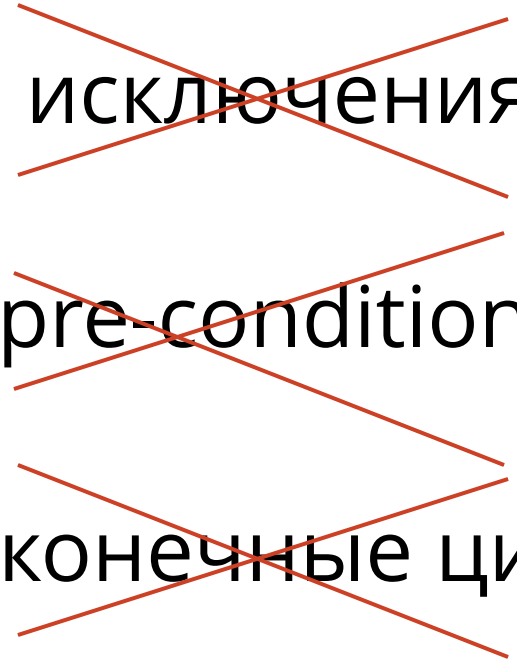
~~исключения~~

~~pre-condition~~

Тотальность

Функция возвращает результат для любого значения входного параметра

за конечное время



Чистота

- одинаковый результат для одинаковых аргументов
- не имеет побочных эффектов

```
def increment(x: Int) = {
  log(x)
  x + 1
}
```

```
def increment(x: Int)(logs: List[String]) =
  (x + 1, x.toString :: logs)
```

```
def increment(x: Int) =
  log(x) as (x + 1)
```

Алгебраические типы данных

```
enum Org{
  case Personnel(name: String, id: Int)
  case Department(name: String)
}
```

Scala 3

```
sealed trait Org

case object Root  extends Org
case class Department(name: String) extends Org
```

Scala 2

Pattern Matching

```
sealed trait Org

case object Root  extends Org
case class Department(name: String) extends Org



def getName(org: Org) = org match {
  case Root                => None
  case Department(name) => Some(name)
}
```

Ещё один ADT

Scala 3

```
enum PersonOrg{
  case Personnel(id: UUID, name: String)
  case Manager(name: String)
}
```

Scala 2

```
sealed trait PersonOrg

case class Personnel(id: UUID, name: String)  extends PersonOrg
case class Manager(name: String) extends PersonOrg
```

Комбинация

```
type ExtendedOrg = Either[Org, PersonOrg]
```

Scala 3

Scala 2

Комбинация

```
type ExtendedOrg = Either[Org, PersonOrg]
```

ИЛИ

Scala 3

```
enum ExtOrg{
  case Simple(org: Org)
  case Person(personOrg: PersonOrg)
}
```

Scala 2

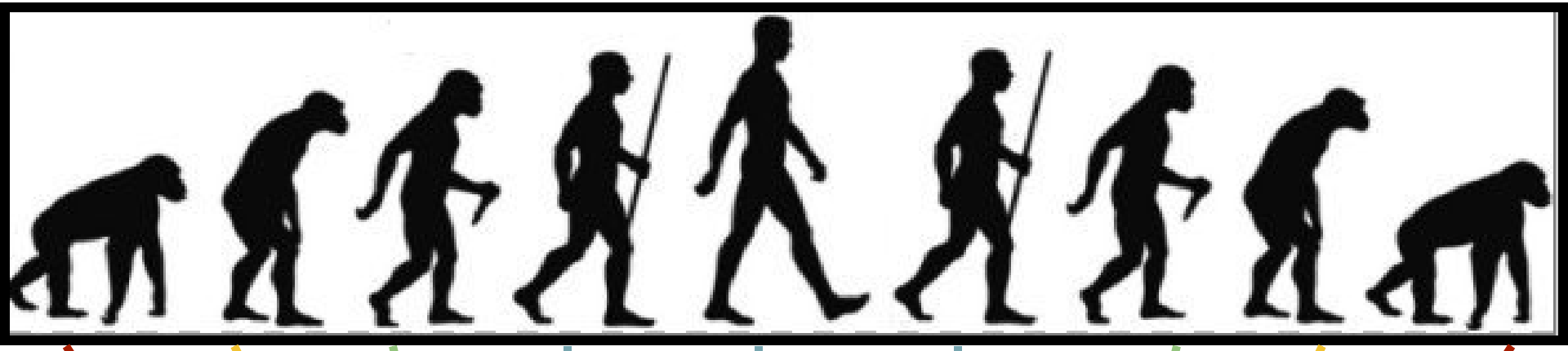
```
sealed trait ExtOrg

case class Simple(org: Org) extends ExtOrg
case class Person(personOrg: PersonOrg) extends ExtOrg
```

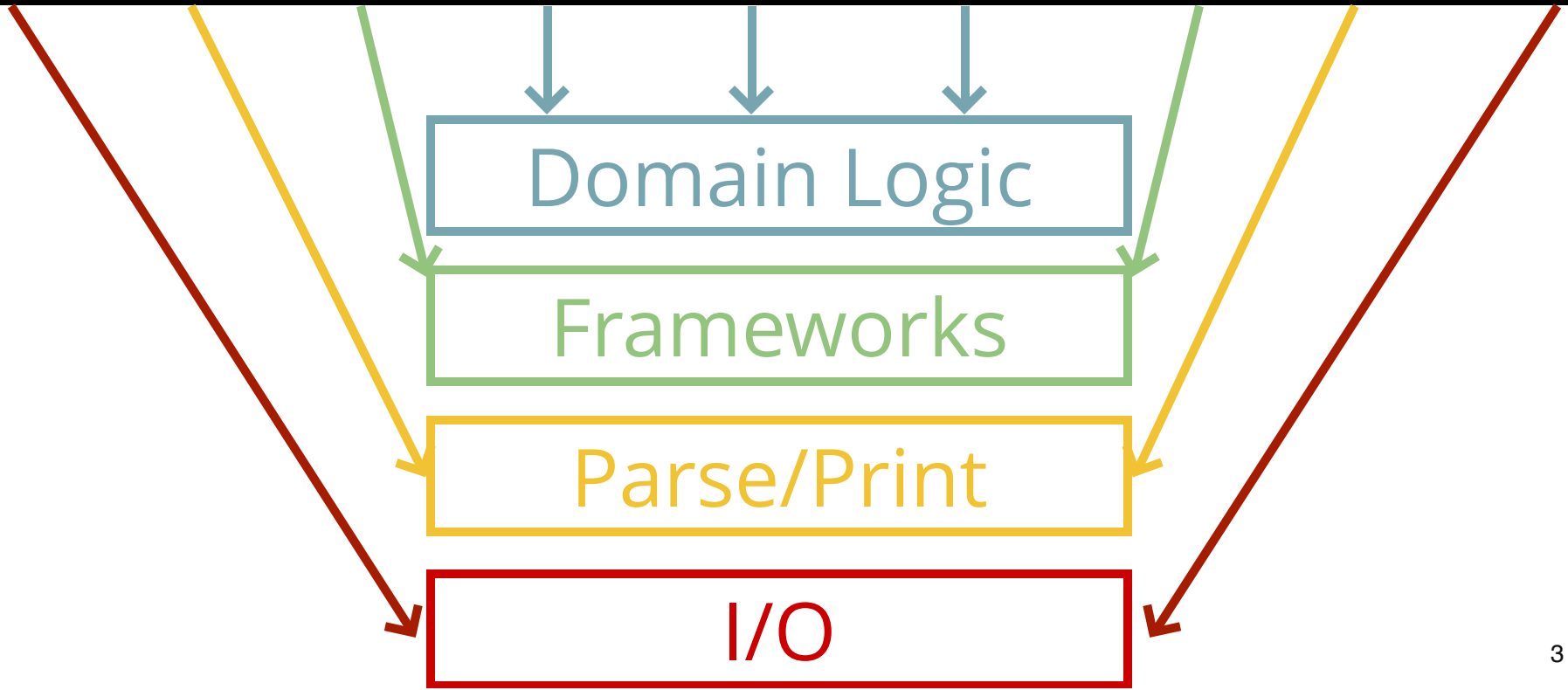
Комбинация

```
sealed trait ExtOrg

case class Simple(org: Org) extends ExtOrg
case class Person(personOrg: PersonOrg) extends ExtOrg
```

```
def getName(org: ExtOrg): Option[String] =
  org match {
    case Simple(Department(name))   => Some(name)
    case Simple(Root)               => None
    case Person(Personnel(_, name)) => Some(name)
    case Person(Manager(name))      => Some(name)
  }
```

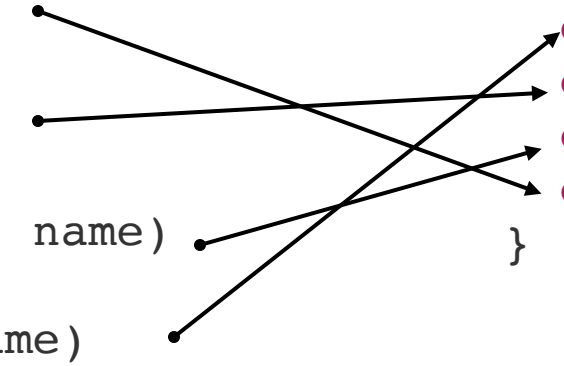
Эволюция данных





Транформации

```
def readOrg(...): Org = {
  ...
  Manager(name)
  ...
  Root
  ...
  Personnel(id, name)
  ...
  Department(name)

}
```

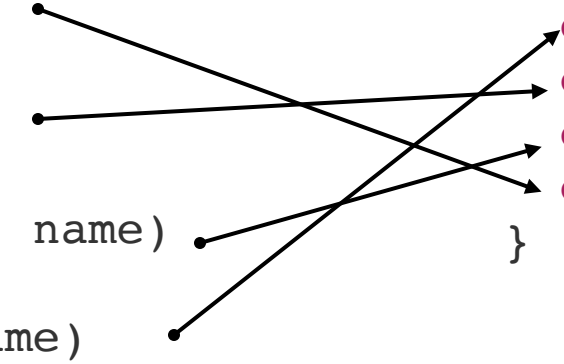


```
def def getName(org: Org) =
  org match {
    case Department(name)    =>
    case Root                =>
    case Personnel(_, name)  =>
    case Manager(name)       =>
    }
```

Транформации

```
def readOrg(...): Org = {
  ...
  Manager(name)
  ...
  Root
  ...
  Personnel(id, name)
  ...
  Department(name)

}
```

```
def def getName(org: Org) =
  org match {
    case Department(name)    =>
    case Root                =>
    case Personnel(_, name)  =>
    case Manager(name)       =>
  }
```

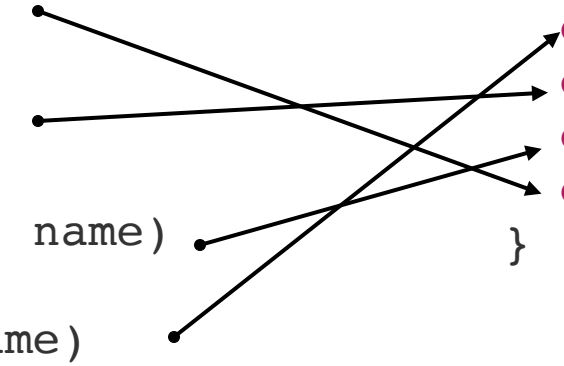


Allocation

Транформации

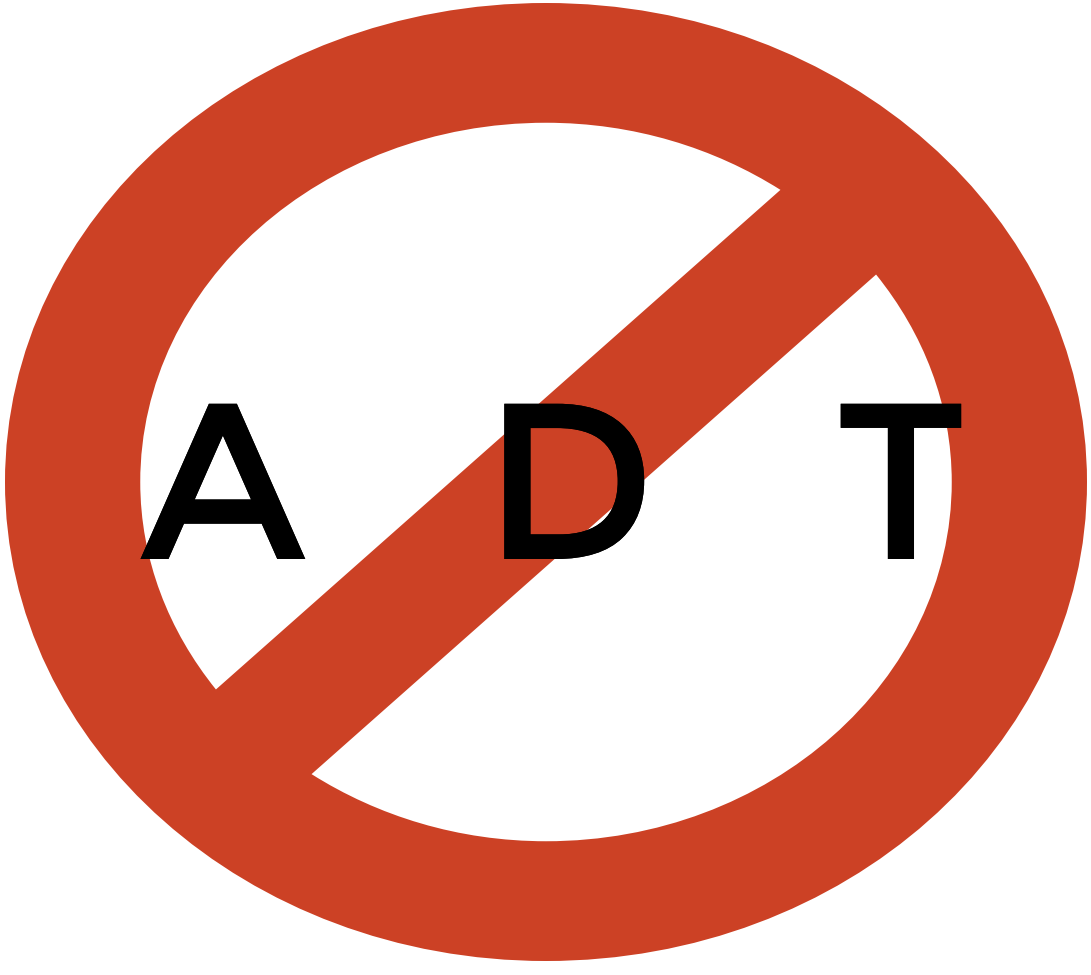
```
def readOrg(...): Org = {
  ...
  Manager(name)
  ...
  Root
  ...
  Personnel(id, name)
  ...
  Department(name)

}
```

```
def def getName(org: Org) =
  org match {
    case Department(name)    =>
    case Root                =>
    case Personnel(_, name)  =>
    case Manager(name)       =>
  }
```



Allocation

Branching


A D T

Категории

Категории

Объекты: A, B, C

Категории

Объекты: A, B, C Типы

Категории

Объекты: **A**, **B**, **C** Типы

Морфизмы: **A** $\longrightarrow$ **B**

Категории

Объекты: **A, B, C** Типы

Морфизмы: **A → B** Функции

Категории

Объекты: A, B, C — Типы

Морфизмы: $A \to B$ — Функции

Identity:

$id[A] : A \to A$

Категории

Объекты: A, B, C — Типы

Морфизмы: $A \to B$ — Функции

Identity:

$id[A] : A \to A$

Композиция:

$f: A \to B$

$g: B \to C$

$g \cdot f : A \to C$

Категории

$$f : A \to B,\ g: B \to C,\ h: C \to D$$

$$id[B] \cdot f = f$$

$$f \cdot id[A] = f$$

$$h \cdot (g \cdot f) = (h \cdot g) \cdot f$$

Изоморфизм

$$A \cong B$$

$$f: A \to B$$

$$g: B \to A$$

$$f \cdot g = id[B]$$

$$g \cdot f = id[A]$$

Изоморфизм

String ≅ List[Char]

```
def stringToList(s: String): List[Char] =  s.toList

def listToString(l: List[Char]): String =  l.mkString
```

Произведение

$$A \times B$$

$$\pi_1 : A \times B \to A, \pi_2 : A \times B \to B$$

$$f: A \to B , g: A \to C$$

$$\langle f,g \rangle: A \to B \times C$$

Произведение

$$A_1 \times A_2 \times \ldots A\square$$

$$\pi\square : A_1 \times A_2 \times \ldots A\square \rightarrow A\square$$

$$f_1: A \rightarrow B_1 , f_2: A \rightarrow B_2, \ldots , f\square: A \rightarrow B\square$$

$$\langle f_1, f_2, \ldots f\square \rangle: A \rightarrow B_1 \times B_2 \times \ldots B\square$$

Произведение
case class \ Tuple

```scala
case class User(name: String, age: Int, sex: Sex)

type UserT = (String, Int, Sex)
```

```scala
val name: String
val age: Int
val sex: Sex

val user: User = User(name, age, sex)

user.name

val userT: UserT = (name, age, sex)

userT._1
```

Экспоненциал

$$B^A$$

$$f: A \times B \to C$$

$$f': A \to C^B$$

$$A \times B^A \to B$$

Экспоненциал

```
type I2C = Int => Char
```

Экспоненциал

```
type I2C = Int => Char
```

Char$^{\text{Int}}$

```
def charAt(s: String, index: Int): Char =
  s(index % s.length)

def charAtCurried(s: String): Int => Char =
  index => s(index % s.length)
```

Экспоненциал

```
type I2C = Int => Char
```

Char Int

```
def charAt(s: String, index: Int): Char =
  s(index % s.length)

def charAtCurried(s: String): Int => Char =
  index => s(index % s.length)
```

```
val i2c: Int => Char

val index = 10

i2c(index)
```

Экспоненциал произведения

$$A^{B_1 \times B_2 \times \ldots B\square}$$

Тип функций нескольких параметров

```
type MkUser = (Int, String) => User
```

Произведение экспоненциалов

$$A_1^{B_1} \times A_2^{B_2} \times \ldots A\square^{B\square}$$

trait

```
type OrgNodeT =  (
    () => String,
    Int => String,
    (String, Int) => Int
)
```

Произведение экспоненциалов

$$A_1^{B_1} \times A_2^{B_2} \times \ldots A\square^{B\square}$$

trait

```
type OrgNodeT =  (
    () => String,
    Int => String,
    (String, Int) => Int
)


trait OrgNode {
    def name: String
    def parent(level: Int): String
    def commonPers(another: String, depth: Int): Int
}
```

Сумма

A + B

$$i_1 : A \to A + B,\ i_2 : B \to A + B$$

$$f: A \to B\ ,\ g: C \to B$$

$$f + g: A + C \to B$$

Сумма

$$A_1 + A_2 + \dots A\square$$

$$i\square : \; A\square \rightarrow A_1 + A_2 + \dots A\square$$

$$f_1 : A_1 \rightarrow B \; , \; f_2 : A_2 \rightarrow B, \dots , \; f\square : A\square \rightarrow B$$

$$f_1 + f_2 + \dots f\square : A_1 + A_2 + \dots A\square \rightarrow B$$

Сумма

```
Either[A, B]
```

```
sealed trait Org

case object Root                                  extends Org
case class Department(name: String)               extends Org
case class Personnel(id: UUID, name: String) extends Org
```

$$\text{Org} \cong \text{Unit} + \text{String} + (\text{UUID} \times \text{String})$$

pattern matching

$$A^{B} \times A^{C} \cong A^{B+C}$$

pattern matching

$$A^{B} \times A^{C} \cong A^{B+C}$$

$$A^{B_1} \times A^{B_2} \times \ldots A^{B\square} \cong A^{B_1 + B_2 + B\square}$$

pattern matching

```
sealed trait Org

case object Root                                  extends Org
case class Department(name: String)               extends Org
case class Personnel(id: UUID, name: String) extends Org
```

pattern matching

```
sealed trait Org

case object Root                                  extends Org
case class Department(name: String)               extends Org
case class Personnel(id: UUID, name: String) extends Org

Org => String
```

pattern matching

```
sealed trait Org

case object Root                                 extends Org
case class Department(name: String)              extends Org
case class Personnel(id: UUID, name: String) extends Org

Org => String


(Unit + String  + (UUID × String)) => String
```

pattern matching

```
sealed trait Org

case object Root                                  extends Org
case class Department(name: String)               extends Org
case class Personnel(id: UUID, name: String) extends Org

Org => String


(Unit + String  + (UUID × String)) => String


(Unit => String) × (String => String)  × (UUID × String => String)
```

pattern matching

```
sealed trait Org

case object Root                                  extends Org
case class Department(name: String)               extends Org
case class Personnel(id: UUID, name: String) extends Org

Org => String


(Unit + String  + (UUID × String)) => String


(Unit => String) × (String => String)  × (UUID × String => String)


trait OrgToString {
  def root: String
  def department(count: Int): String
  def personnel(id: UUID, name: String): String
}
```

pattern matching

```
type OrgToSgring = Org => A

trait OrgToString {
  def root: String
  def department(count: Int): String
  def personnel(id: UUID, name: String): String
}
```

```
type OrgTo[A] = Org => A

trait OrgTo[A] {
  def root: A
  def department(count: Int): A
  def personnel(id: UUID, name: String): A
}
```

Продолжения

$$A \cong \int_X X^{(X^A)}$$

Продолжения

```
trait Cont[A]{
  def continue[X](k: A => X): X
}
```

$$\mathsf{Cont}[A] \cong A$$

Продолжения

```
trait Cont[A]{
  def continue[X](k: A => X): X
}
```

Cont[A] ≅ A

```
def fromCont(cont: Cont[A]): A =
    cont.continue(x => x)

def toCont(a: A): Cont[A] =
    new Cont[A]{
       def continue[X](f: A => X): X = f(a)
    }
```

pattern matching

```
sealed trait Org

case object Root                                  extends Org
case class Department(name: String)               extends Org
case class Personnel(id: UUID, name: String) extends Org
```

pattern matching

```
sealed trait Org

case object Root                                  extends Org
case class Department(name: String)               extends Org
case class Personnel(id: UUID, name: String) extends Org

trait OrgTo[A] {
  def root: A
  def department(count: Int): A
  def personnel(id: UUID, name: String): A
}
```

pattern matching

```
sealed trait Org

case object Root                                  extends Org
case class Department(name: String)               extends Org
case class Personnel(id: UUID, name: String) extends Org

trait OrgTo[A] {
  def root: A
  def department(count: Int): A
  def personnel(id: UUID, name: String): A
}


trait OrgCont{
  def continue[A](k: OrgTo[A]): A
}
```

pattern matching

```
sealed trait Org

case object Root                                  extends Org
case class Department(name: String)               extends Org
case class Personnel(id: UUID, name: String) extends Org

trait OrgTo[A] {
  def root: A
  def department(count: Int): A
  def personnel(id: UUID, name: String): A
}


trait OrgCont{
  def continue[A](k: OrgTo[A]): A
}
```

Org ≅ OrgCont

pattern matching

```
def createOrg(tag: Int, name: String): Org =
  tag match {
   case 1 => Root
   case 2 => Personel(name, UUID.randomUUID())
   case 3 => Department(name)
}
```

pattern matching

```
def createOrg(tag: Int, name: String): Org =
  tag match {
   case 1 => Root
   case 2 => Personel(name, UUID.randomUUID())
   case 3 => Department(name)
}
```

```
def createOrg[A](tag: Int, name: String)(org: OrgTo[A]): A =
    tag match {
       case 1 => org.root
       case 2 => org.personel(name, UUID.randomUUID())
       case 3 => org.department(name)
    }
```

КОМПОЗИЦИЯ

КОМПОЗИЦИЯ

```
trait ManagerTo[A]{
  def manager(name: String): A
}
```

```
def createOrg[A](tag: Int, name: String)
                (org: OrgTo[A], man: ManagerTo[A]): A =
    tag match {
       case 1 => org.root
       case 2 => org.personel(name, UUID.randomUUID())
       case 3 => org.department(name)
       case 4 => man.manager(name)
    }
```

Рекурсивный ADT

```
sealed trait Org

case class Pers(name: String, orgId: Long) extends Org

case class Dept(name: String, sub: List[Org]) extends Org
```

fixpoint

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

fixpoint

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


def fact1(n: Int)(recur: => Int => Int) =
  if (n == 0) 1
  else n * recur(n - 1)
```

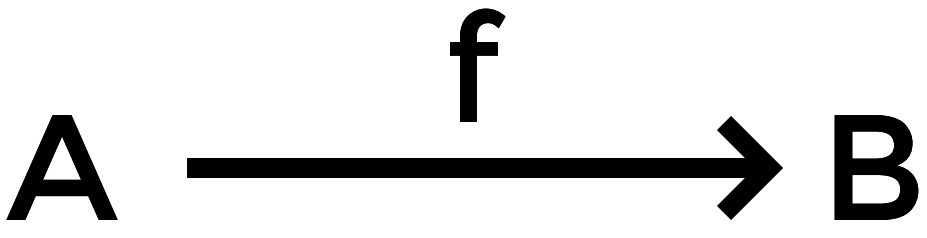
fixpoint

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


def fact1(n: Int)(recur: => Int => Int) =
  if (n == 0) 1
  else n * recur(n - 1)

def fix[A](f: (=> A) => A): A = f(fix(f))
```

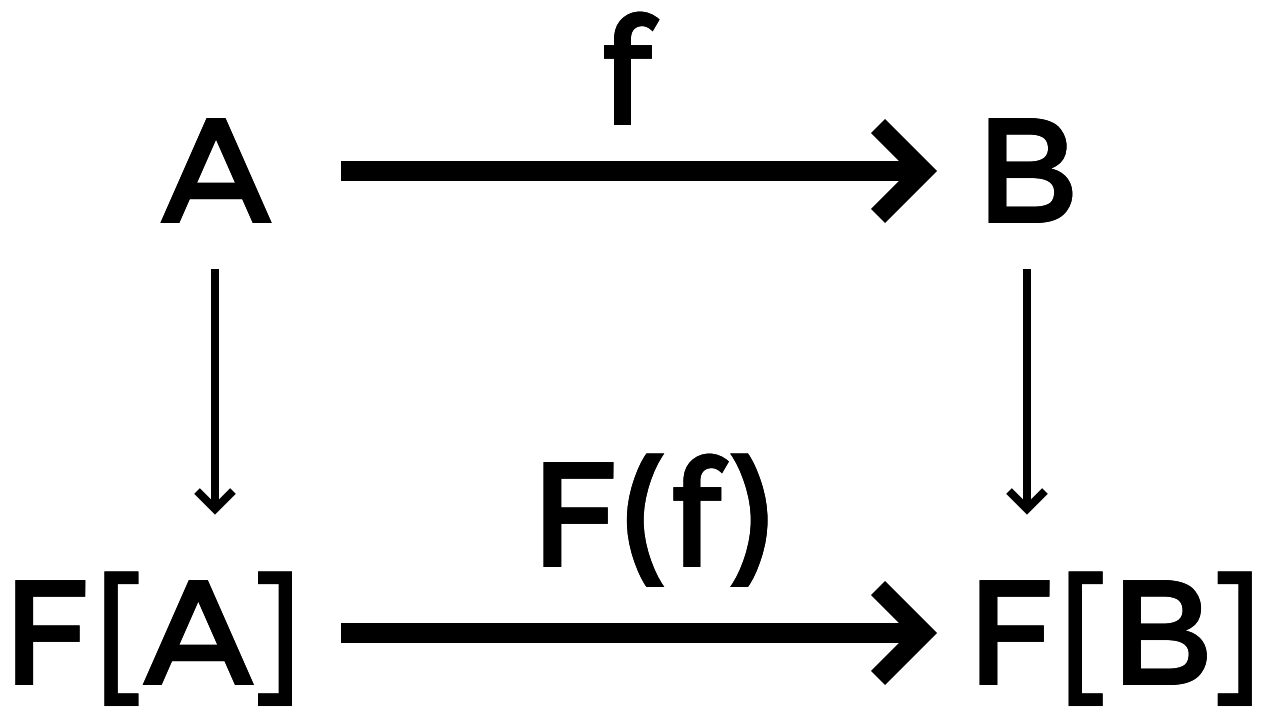
fixpoint

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


def fact1(n: Int)(recur: => Int => Int) =
  if (n == 0) 1
  else n * recur(n - 1)

def fix[A](f: (=> A) => A): A = f(fix(f))


val factFixed = fix(fact1)
```

Функтор

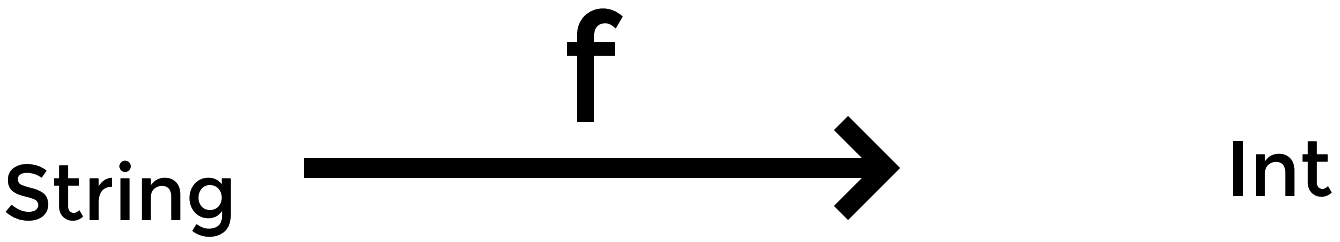


Функтор



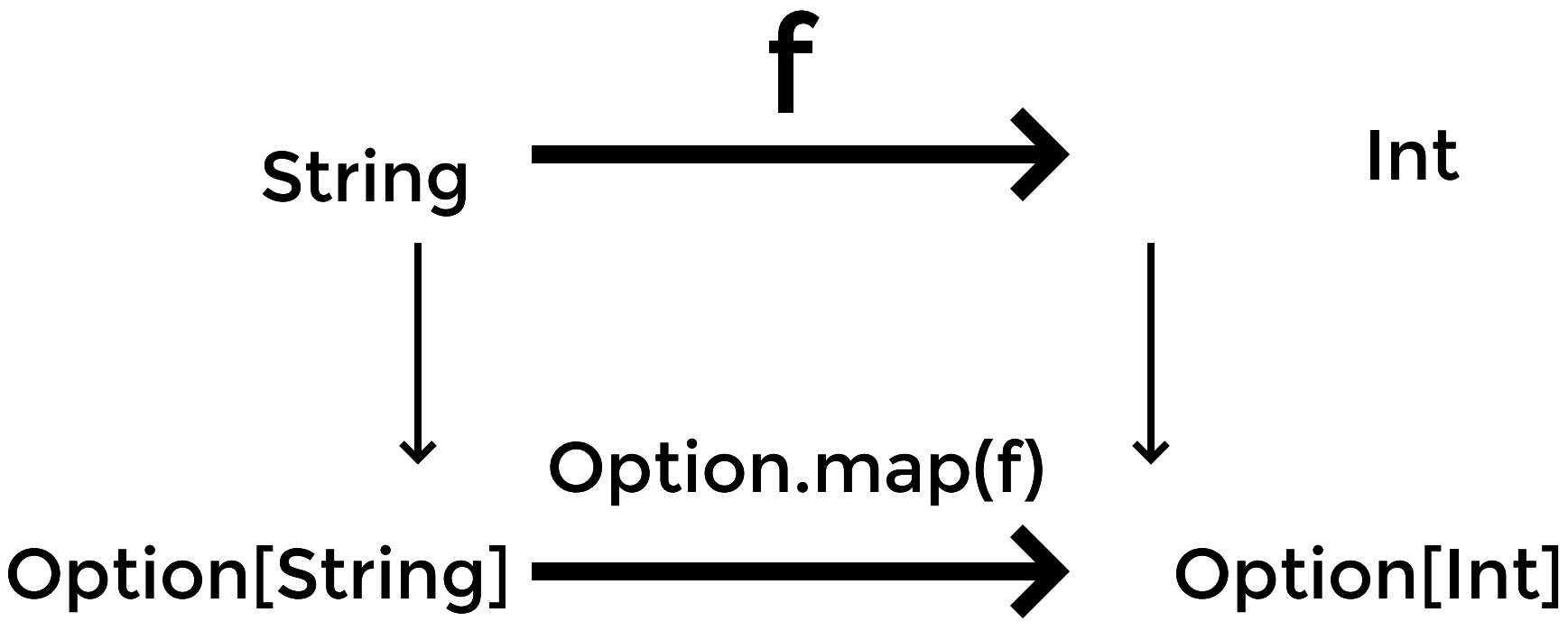
$F(f \cdot g) = F(f) \cdot F(g)$ $F(id) = id$

Функтор




```
def map[A, B](f: A => B): F[A] => F[B]
```

Функтор




```
def map[A, B](f: A => B): F[A] => F[B]
```

Функтор

```
trait Functor[F[_]]{
    def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

```
map(fa)(identity) == identity

map(fa)(f compose g) == map(map(fa)(g))(f)
```

Option List

Future Observable

Function Id

Tuple Either

Функтор

```
sealed trait OrgF[A]

case class PersF[A](name: String, orgId: Long) extends OrgF[A]
case class DeptF[A](name: String, sub: List[A]) extends OrgF[A]
```

Функтор

```
sealed trait OrgF[A]

case class PersF[A](name: String, orgId: Long) extends OrgF[A]
case class DeptF[A](name: String, sub: List[A]) extends OrgF[A]

val orgFunctor = new Functor[OrgF]{
  override def map[A, B](fa: OrgF[A])(f: A => B) =
    fa match {
      case PersF(name, orgId) => PersF(name, orgId)
      case DeptF(name, sub) => DeptF(name, sub.map(f))
    }
}
```

Функтор

```
sealed trait OrgF[A]

case class PersF[A](name: String, orgId: Long) extends OrgF[A]
case class DeptF[A](name: String, sub: List[A]) extends OrgF[A]

val orgFunctor = new Functor[OrgF]{
  override def map[A, B](fa: OrgF[A])(f: A => B) =
    fa match {
      case PersF(name, orgId) => PersF(name, orgId)
      case DeptF(name, sub) => DeptF(name, sub.map(f))
    }
}
```

```
type Org = OrgF[OrgF[OrgF[...]]]
```

Функтор

```
case class Fix[F[_]](value: F[Fix[F]]
```

```
Org ≅ Fix[OrgF]
```

Алгебра

```
trait Monoid[A]{
 def combine(x: A, y: A): A
 def empty: A
}


 trait LinSpace[A]{
   def plus(x: A, y: A): A
   def times(k: Double, x: A): A
 }
```

Алгебра

```
trait Monoid[A]{

 def combine(x: A, y: A): A

 def empty: A

}
```

```
sealed trait MonoidF[A]

case class Combine(x: A, y: A)
   extends MonoidF
case object Empty
   extends MonoidF
```

```
trait LinSpace[A]{

  def plus(x: A, y: A): A

  def times(k: Double, x: A): A

}
```

```
sealed trait LinSpaceF[A]

case class Plus(x: A, y: A)
   extends LinSpaceF[A]
case class Times(k: Double, x: A)
   extends LinSpaceF[A]
```

Алгебра

```
trait Monoid[A]{

 def combine(x: A, y: A): A

 def empty: A

}
```

```
sealed trait MonoidF[A]

case class Combine(x: A, y: A)
   extends MonoidF
case object Empty
   extends MonoidF
```

Monoid[A] ≅ MonoidF[A] => A

```
trait LinSpace[A]{

  def plus(x: A, y: A): A

  def times(k: Double, x: A): A

}
```

```
sealed trait LinSpaceF[A]

case class Plus(x: A, y: A)
   extends LinSpaceF[A]
case class Times(k: Double, x: A)
   extends LinSpaceF[A]
```

LinSpace[A] ≅ LinSpaceF[A] => A

Алгебра Эндофунктора

$$F[A] \to A$$

Алгебра

```
sealed trait Org
case class Pers(name: String, orgId: Long) extends Org
case class Dept(name: String, left: Org, right: Org) extends Org
```

Алгебра

```
sealed trait Org
case class Pers(name: String, orgId: Long) extends Org
case class Dept(name: String, left: Org, right: Org) extends Org
```

```
sealed trait OrgF[A]
case class PersF[A](name: String, orgId: Long) extends OrgF[A]
case class DeptF[A](name: String, sub: List[A]) extends OrgF[A]
```

Алгебра

```
sealed trait Org
case class Pers(name: String, orgId: Long) extends Org
case class Dept(name: String, left: Org, right: Org) extends Org
```

```
sealed trait OrgF[A]
case class PersF[A](name: String, orgId: Long) extends OrgF[A]
case class DeptF[A](name: String, sub: List[A]) extends OrgF[A]
```

```
type OrgAlgebra[A] = OrgF[A] => A
```

Алгебра

```
sealed trait Org
case class Pers(name: String, orgId: Long) extends Org
case class Dept(name: String, left: Org, right: Org) extends Org
```

```
sealed trait OrgF[A]
case class PersF[A](name: String, orgId: Long) extends OrgF[A]
case class DeptF[A](name: String, sub: List[A]) extends OrgF[A]
```

```
type OrgAlgebra[A] = OrgF[A] => A
```

```
trait OrgAlgebra[A]{
  def pers(name: String, orgId: Long): A
  def dept(name: String, sub: List[A]): A
}
```

Алгебры

```
trait OrgAlgebra[A]{
  def pers(name: String, orgId: Long): A
  def dept(name: String, left: A, right: A): A
}


object Size extends PersAlgebra[Int]{
  def pers(name: String, orgId: Long) = 1
  def dept(name: String, sub: List[Int]) = sub.sum
}


object Names extends OrgAlgebra[List[String]]{
  def pers(name: String, orgId: Long) = List(name)
  def dept(name: String, sub: List[List[String]]) = sub.flatten
}
```

Категория Алгебр

Объекты: $F[A] \to A$

Морфизмы: $A \to B$

$$X: F[A] \to A$$

$$Y: F[B] \to B$$

$$f: A \to B$$

$$f \cdot X = Y \cdot F(f)$$

Категория Алгебр

```
object Size extends PersAlgebra[Int]{
  def pers(name: String, orgId: Long) = 1
  def dept(name: String, sub: List[Int]) = sub.sum
}
```

```
object Names extends OrgAlgebra[List[String]]{
  def pers(name: String, orgId: Long) = List(name)
  def dept(name: String, sub: List[List[String]]) = sub.flatten
}
```

```
val count: List[String] => Int = _.length
```

Категория Алгебр

```
count(
   Names.dept(name1, List(
     Names.dept(name2, List(
        Names.pers(name3, id3),
        Names.pers(name4, id4 )),
   Names.pers(name5, id5)))
```

```
Size.dept(name1, List(
   count(
      Names.dept(name2, List(
        Names.pers(name3, id3),
        Names.pers(name4, id4 )),
   count(Name.pers(name5, id5)))))
```

Категория Алгебр

```
Size.dept(name1, List(
      Size.dept(name2, List(
        count(Names.pers(name3, id3)),
        count(Names.pers(name4, id4))),
      Size.pers(name5, id5)))
```

```
Size.dept(name1, List(
      Size.dept(name2, List(
        Size.pers(name3, id3),
        Size.pers(name4, id4)),
      Size.pers(name5, id5)))
```

Категория Алгебр

```
trait OrgAlgebra[A]{
  def pers(name: String, orgId: Long): A
  def dept(name: String, sub: List[A]): A
}
```

```
val algA: OrgAlgebra[A]
val algB: OrgAlgebra[B]

val f: A => B

f(algA.pers(name, id)) == algB.pers(name, id)
f(algA.dept(name, subs)) == algB.dept(name, subs.map(f))
```

Начальная алгебра

$$F[\mu[F]] \to \mu[F]$$

Начальная алгебра

$$F[\mu[F]] \rightarrow \mu[F]$$

$$X\colon F[A] \rightarrow A$$

$$I(X)\colon \mu[F] \rightarrow A$$

Начальная алгебра

$$F[\mu[F]] \to \mu[F]$$

$$X\colon F[A] \to A$$

$$I(X)\colon \mu[F] \to A$$

$$\mu[F] \cong F[\mu[F]]$$

Начальная алгебра

$$F[\mu[F]] \rightarrow \mu[F]$$

$$X\colon F[A] \rightarrow A$$

$$I(X)\colon \mu[F] \rightarrow A$$

$$\mu[F] \cong F[\mu[F]]$$

$$\mu[F] \cong F[\mu[F]] \cong F[F[\mu[F]]] \cong F[F[F[\ldots]]]$$

Начальная алгебра

$$F[\mu[F]] \to \mu[F]$$

$$X\colon F[A] \to A$$

$$I(X)\colon \mu[F] \to A$$

$$\mu[F] \cong F[\mu[F]]$$

$$\mu[F] \cong F[\mu[F]] \cong F[F[\mu[F]]] \cong F[F[F[\ldots]]]$$

$$\mu[\mathbf{OrgF}] \cong \mathbf{Org}$$

Начальная алгебра

```
def orgI[A]: OrgAlgebra[A] => Org => A
```

```
def orgI[A]: OrgAlgebra[A] => Org => A = alg => {
  case Pers(name, orgId) => alg.pers(name, orgId)
  case Dept(name, sub)   => alg.dept(name, sub.map(orgI(alg)))
}
```

Начальная алгебра

```
def orgI[A]: OrgAlgebra[A] => Org => A
```

```
def orgI[A]: OrgAlgebra[A] => Org => A = alg => {
  case Pers(name, orgId) => alg.pers(name, orgId)
  case Dept(name, sub)   => alg.dept(name, sub.map(orgI(alg)))
}
```

```
def foldOrg[A]: Org => OrgAlgebra[A] => A
```

Начальная алгебра

```
def orgI[A]: OrgAlgebra[A] => Org => A
```

```
def orgI[A]: OrgAlgebra[A] => Org => A = alg => {
  case Pers(name, orgId) => alg.pers(name, orgId)
  case Dept(name, sub)   => alg.dept(name, sub.map(orgI(alg)))
}
```

```
def foldOrg[A]: Org => OrgAlgebra[A] => A
```

```
object OrgOrg extends OrgAlgebra[Org]{
  def pers(name: String, orgId: Long) = Pers(name, orgid)
  def dept(name: String, sub: List[Org]]) = Dept(name, sub)
}
```

Начальная алгебра

```
trait OrgInit{
  def fold[A](algebra: OrgAlgebra[A]): A
}
```

`Org ≅ OrgInit`

Конечное представление

```
def makeOrg(): Org =
   Dept(name1,
     List(
       Pers(name2, id2),
       Dept(name3, List())))
```

```
def makeOrg[A]()(org: OrgAlgebra[A]): A =
   org.dept(name1,
     List(
       org.pers(name2, id2),
       org.dept(name3, List())))
```

Композиция

```
trait ManAlgebra[A]{
  def manager(name: String, sub: A): A
}
```

```
def makeOrg[A](...)(org: OrgAlgebra[A], man: ManAlgebra[A]): A =
   org.dept(name1,
     List(
       org.pers(name2, id2),
       man.manager(name,
         org.dept(name3, List()))))
```

Неявные параметры

```
def makeOrg[A: OrgAlgebra : ManAgebra](...): A =
    dept(name1,
      List(
        pers(name2, id2),
        manager(name,
          dept(name3, List()))))
```

Tagless final

```
sealed trait Data
case class Variant
foo: Data
match {...}
```

```
trait Data[A]
def Variant
def foo[A: Data]: A
new Data{...}
```

Типизированные алгебры

```
trait Org[F[_]]{
   def root(dept: F[DeptId]): F[Root]
   def personnel(id: UUID, name: String): F[UserId]
   def manager(name: String, dept: F[DeptId]): F[PersId]
   def department(name: String,
                  sub: List[F[DeptId]],
                  pers: List[F[PersId]]): F[DeptId]
}
```

Типизированные алгебры

```
trait Org[F[_]]{
   type DeptId
   type UserId
   type Root

   def root(dept: F[DeptId]): F[Root]
   def personnel(id: UUID, name: String): F[UserId]
   def manager(name: String, dept: F[DeptId]): F[PersId]
   def department(name: String,
                  sub: List[F[DeptId]],
                  pers: List[F[PersId]]): F[DeptId]
}
```

Типизированные алгебры

```
trait Org[F[_]]{
   def makeRoot(dept: DeptId): F[Root]
   def makePersonnel(id: UUID, name: String): F[UserId]
   def makeManager(name: String, dept: DeptId): F[PersId]
   def makeDepartment(name: String,
                      sub: List[DeptId],
                      pers: List[PersId]): F[DeptId]
}
```

Монады

```
trait Monad[F[_]] extends Functor[F]{
   def pure[A](a: A): F[A]
   def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

Option List

Reader Id

Writer Either

State

IO

Монады

```
def createOrg[F[_]: Monad: Org]: F[Root] = for{
   pers1 <- makePersonnel(uuid1, name1)
   pers1 <- makePersonnel(uuid2, name2)
   personnels = List(pers1, pers2)
   dept  <- makeDept(name3, personnels)
   root  <- makeRoot(dept)
} yield root
```

Эффекты

```
def createOrg(): RootId
```

Грязь

Эффекты

```
def createOrg(): RootId
```

Грязь

```
def createOrg(): IO[RootId]
```

Неизвестность

Эффекты

```
def createOrg(): RootId
```

Грязь

```
def createOrg(): IO[RootId]
```

Неизвестность

```
def createOrg[F[_] : Monad
                   : Org
                   : Raise[?[_], OrgError]
                   : DataRead[?[_], HR]
                   ](): F[RootId]
```

Блаженство

Tagless final

Алгебра призвана
выполнять ровно
одну задачу в рамках
одного эффекта

Tagless final

Алгебра призвана
выполнять ровно
одну задачу в рамках
одного эффекта

Single Responsibility

Tagless final

Композиция алгебр проста и естественна, как добавлением параметра в метод

Tagless final

Композиция алгебр проста и естественна, как добавлением параметра в метод

Open - Closed

Tagless final

Функции с более слабыми требованиями к алгебрам всегда применимы в более сильном контексте

Tagless final

Функции с более слабыми требованиями к алгебрам всегда применимы в более сильном контексте

Liskov substitution

Tagless final

Алгебры склонны быть настолько слабыми, насколько возможно для простейшего функционала

Tagless final

Алгебры склонны быть настолько слабыми, насколько возможно для простейшего функционала

Interface segregation

Tagless final

Алгебры реализуют косвенную полиморфную зависимость функций от требований

Tagless final

Алгебры реализуют косвенную полиморфную зависимость функций от требований

Dependency Inversion

Tagless final

FP is the best OOP

Tagless final

TF

~~FP~~ is the best OOP

Что ещё посмотреть

1. Edmund Noble

   Data, and when not to use it

   https://www.youtube.com/watch?v=q6JCvdMWtmo

2. Adam Warski

   Free monad or tagless final? How not to commit to a monad too early

   https://www.youtube.com/watch?v=IhVdU4Xiz2U

Вопросы

email : odomontois@gmail.com
o.nizhnikov@tinkoff.ru

telegram : https://t.me/odomontois