Скалисты уже знают зависимые типы

Но это не точно

Денис Буздалóв

23 ноября 2023

Зависимые типы?

Зависимые типы?

- Что?

Зависимые типы?

- Что?
- В Scala?

Зависимые типы?

- Что?
- В Scala?
- Зачем может быть нужно?

Зависимые типы?

- Что?
- В Scala?
- Зачем может быть нужно?
- Beyond Scala

Дисклеймеры

Дисклеймеры

- Лекция познавательно-развлекательная

Дисклеймеры

- Лекция познавательно-развлекательная
- Слушатели, возможно, Scala знают лучше докладчика

Дисклеймеры

- Лекция познавательно-развлекательная
- Слушатели, возможно, Scala знают лучше докладчика
- Слайды с последовательно появляющимся текстом

Дисклеймеры

- Лекция познавательно-развлекательная
- Слушатели, возможно, Scala знают лучше докладчика
- Слайды с последовательно появляющимся текстом
- Некоторые вещи, для упрощения изложения, будут заметаться под ковёр

О лекции

- Будет много кода

О лекции

- Будет много кода
- Многое будет иллюстрироваться в непривычном синтаксисе

О лекции

- Будет много кода
- Многое будет иллюстрироваться в непривычном синтаксисе
- Idris — довольно произвольный выбор

О лекции

- Будет много кода
- Многое будет иллюстрироваться в непривычном синтаксисе
- Idris — довольно произвольный выбор
  - полноценные нативные зависимые типы
  - достаточно практичный
  - докладчик банально лучше всего с ним знаком ;-)

О лекции

- Будет много кода
- Многое будет иллюстрироваться в непривычном синтаксисе
- Idris — довольно произвольный выбор
  - полноценные нативные зависимые типы
  - достаточно практичный
  - докладчик банально лучше всего с ним знаком ;-)
- Цель — понимание концепций слушателями, поэтому вопросы, уточнения в процессе

О лекции

- Будет много кода
- Многое будет иллюстрироваться в непривычном синтаксисе
- Idris — довольно произвольный выбор
  - полноценные нативные зависимые типы
  - достаточно практичный
  - докладчик банально лучше всего с ним знаком ;-)
- Цель — понимание концепций слушателями, поэтому вопросы, уточнения в процессе
- Нету цели унизить Скалу, ни синтаксически, ни в теории типов

О лекции

- Будет много кода
- Многое будет иллюстрироваться в непривычном синтаксисе
- Idris — довольно произвольный выбор
  - полноценные нативные зависимые типы
  - достаточно практичный
  - докладчик банально лучше всего с ним знаком ;-)
- Цель — понимание концепций слушателями, поэтому вопросы, уточнения в процессе
- Нету цели унизить Скалу, ни синтаксически, ни в теории типов
- Привыкание, зависимость, во все тяжкие...

Синтаксис

Синтаксис

```
enum Tree[A]:
  case Leaf(a: A)
  case Node(value: A, to: NonEmptyList[Tree[A]])
```

Синтаксис

```
enum Tree[A]:
  case Leaf(a: A)
  case Node(value: A, to: NonEmptyList[Tree[A]])
```

```
data Tree a
  = Leaf a
  | Node a (List1 (Tree a))
```

Синтаксис

```
given Functor[Tree] with
  def map[A, B](fa: Tree[A])(f: A ⇒ B): Tree[B] =
    def go(tree: Tree[A]): Eval[Tree[B]] = tree match
      case Leaf(x) ⇒ Eval.now(Leaf(f(x)))
      case Node(x, ts) ⇒ Eval.defer(
        ts.traverse(go).map(Node(f(x), _))
      )
    go(fa).value
```

Синтаксис

```
given Functor[Tree] with
  def map[A, B](fa: Tree[A])(f: A ⇒ B): Tree[B] =
    def go(tree: Tree[A]): Eval[Tree[B]] = tree match
      case Leaf(x) ⇒ Eval.now(Leaf(f(x)))
      case Node(x, ts) ⇒ Eval.defer(
        ts.traverse(go).map(Node(f(x), _))
      )
    go(fa).value
```

```
Functor Tree where
  map f = eval . go where
    go : Tree a -> Eval $ Tree b
    go (Leaf x)    = pure $ Leaf $ f x
    go (Node x ts) = defer $ Node (f x) <$> traverse go ts
```

Синтаксис

```
def minimum[A](using Order[A])(tree: Tree[A]): A = tree match
  case Leaf(x)     ⇒ x
  case Node(x, ts) ⇒ x `min` ts.map(minimum).minimum
```

Синтаксис

```scala
def minimum[A](using Order[A])(tree: Tree[A]): A = tree match
  case Leaf(x)     ⇒ x
  case Node(x, ts) ⇒ x `min` ts.map(minimum).minimum
```

```haskell
minimum : Ord a ⇒ Tree a → a
minimum $ Leaf x    = x
minimum $ Node x ts = x `min` minimum (map minimum ts)
```

Синтаксис

```
case class User(
  name: UserName,
  passport: Passport,
  birthDate: BirthDate)

def u : User = ???
def n : UserName = u.name

case class ValidationError(
  msg: String)
```

Синтаксис

```
case class User(
  name: UserName,
  passport: Passport,
  birthDate: BirthDate)

def u : User = ???
def n : UserName = u.name

case class ValidationError(
  msg: String)
```

```
record User where
  constructor MkUser
  name      : UserName
  passport  : Passport
  birthDate : BirthDate

u : User
n : UserName
n = u.name

record ValidationError where
  constructor MkValidationError
  msg : String
```

Синтаксис

```
def validateUserName(input: String):
  ValidatedNel[ValidationError, UserName] = ???
def validatePassport(input: String):
  ValidatedNel[ValidationError, Passport] = ???
def validateBirthDate(input: String):
  ValidatedNel[ValidationError, BirthDate] = ???
```

Синтаксис

```
def validateUserName(input: String):
  ValidatedNel[ValidationError, UserName] = ???
def validatePassport(input: String):
  ValidatedNel[ValidationError, Passport] = ???
def validateBirthDate(input: String):
  ValidatedNel[ValidationError, BirthDate] = ???
```

```
validateUserName :
  String → ValidatedL ValidationError UserName
validatePassport :
  String → ValidatedL ValidationError Passport
validateBirthDate :
  String → ValidatedL ValidationError BirthDate
```

Синтаксис

```
def user: ValidatedNel[ValidationError, User] = (
  validateUserName("Vova"),
  validatePassport("1-1"),
  validateBirthDate("04-02-1942")
).mapN(User.apply)
```

Синтаксис

```
def user: ValidatedNel[ValidationError, User] = (
  validateUserName("Vova"),
  validatePassport("1-1"),
  validateBirthDate("04-02-1942")
).mapN(User.apply)
```

```
user : ValidatedL ValidationError User
user = [| MkUser
            (validateUserName "Vova")
            (validatePassport "1-1")
            (validateBirthDate "04-02-1942")
        |]
```

Старые знакомые?

Старые знакомые?

```
def last[A](xs: List[A]): Option[A] = xs match
  case Nil      ⇒ None
  case x :: Nil ⇒ Some(x)
  case _ :: xs  ⇒ last(xs)
```

Полиморфизм: типы зависят от типов

```
def last[A](xs: List[A]): Option[A] = xs match
  case Nil      ⇒ None
  case x :: Nil ⇒ Some(x)
  case _ :: xs  ⇒ last(xs)
```

Полиморфизм: типы зависят от типов

```scala
def last[A](xs: List[A]): Option[A] = xs match
  case Nil      ⇒ None
  case x :: Nil ⇒ Some(x)
  case _ :: xs  ⇒ last(xs)
```

```haskell
last : List a → Maybe a
last []      = Nothing
last [x]     = Just x
last (x::xs) = last xs
```

Полиморфизм: типы зависят от типов

```scala
def last[A](xs: List[A]): Option[A] = xs match
  case Nil      ⇒ None
  case x :: Nil ⇒ Some(x)
  case _ :: xs  ⇒ last(xs)
```

```haskell
last : forall a. List a → Maybe a
last []     = Nothing
last [x]    = Just x
last (x::xs) = last xs
```

Полиморфизм: типы зависят от типов

```
def last[A](xs: List[A]): Option[A] = xs match
  case Nil      ⇒ None
  case x :: Nil ⇒ Some(x)
  case _ :: xs  ⇒ last(xs)
```

```
last : {0 a : _} → List a → Maybe a
last []      = Nothing
last [x]     = Just x
last (x::xs) = last xs
```

Полиморфизм: типы зависят от типов

```scala
def last[A](xs: List[A]): Option[A] = xs match
  case Nil      ⇒ None
  case x :: Nil ⇒ Some(x)
  case _ :: xs  ⇒ last(xs)
```

```haskell
last : {0 a : Type} → List a → Maybe a
last []      = Nothing
last [x]     = Just x
last (x::xs) = last xs
```

Лёгкая зависимость

Лёгкая зависимость

Пример:

- интерфейс кодировки значения одного типа другим

Лёгкая зависимость

Пример:

- интерфейс кодировки значения одного типа другим
- целевой тип определяется способом кодировки

Лёгкая зависимость

```
trait Encoder[From]:
  type To
  def encode(from: From): To
```

Лёгкая зависимость

```
trait Encoder[From]:
  type To
  def encode(from: From): To

def accEncoded[A](using Dec: Encoder[A])
                 (using Semigroup[Dec.To])
                 (x: A, rest: Dec.To): Dec.To =
  Dec.encode(x) |+| rest
```

Тип зависит от значения

```
trait Encoder[From]:
  type To
  def encode(from: From): To

def accEncoded[A](using Dec: Encoder[A])
                 (using Semigroup[Dec.To])
                 (x: A, rest: Dec.To): Dec.To =
  Dec.encode(x) |+| rest
```

Тип зависит от типа внутри значения

```
trait Encoder[From]:
  type To
  def encode(from: From): To

def accEncoded[A](using Dec: Encoder[A])
                 (using Semigroup[Dec.To])
                 (x: A, rest: Dec.To): Dec.To =
  Dec.encode(x) |+| rest
```

Тип зависит от типа внутри значения

```
trait Encoder[From]:
  type To                                          // <-- type member
  def encode(from: From): To

def accEncoded[A](using Dec: Encoder[A])
                 (using Semigroup[Dec.To])
                 (x: A, rest: Dec.To): Dec.To =
  Dec.encode(x) |+| rest   // ^^^^^^      path-dependent type
```

Тип зависит от типа внутри значения

```
trait Encoder[From]:
  type To                                          // <-- type member
  def encode(from: From): To

def accEncoded[A](using Dec: Encoder[A])
                 (using Semigroup[Dec.To])
                 (x: A, rest: Dec.To): Dec.To =
  Dec.encode(x) |+| rest   // ^^^^^^      path-dependent type
```

```
interface Encoder from where
  0 To : Type
  encode : from -> To
```

Тип зависит от типа внутри значения

```
trait Encoder[From]:
  type To                                          // <-- type member
  def encode(from: From): To

def accEncoded[A](using Dec: Encoder[A])
                 (using Semigroup[Dec.To])
                 (x: A, rest: Dec.To): Dec.To =
  Dec.encode(x) |+| rest   // ^^^^^^      path-dependent type
```

```
interface Encoder from where
  0 To : Type
  encode : from -> To

accEncoded : (dec : Encoder from) => Semigroup (To @{dec}) =>
             from -> To @{dec} -> To @{dec}
accEncoded x rest = encode x <+> rest
```

Тип зависит от типа внутри значения

```
given [A]: Encoder[Tree[A]] with
  type To = List[A]
  def encode(from: Tree[A]): To = toList(from)
```

Тип зависит от типа внутри значения

```
given [A]: Encoder[Tree[A]] with
  type To = List[A]
  def encode(from: Tree[A]): To = toList(from)
```

```
Encoder (Tree a) where
  To = List a
  encode = toList
```

Тип зависит от типа внутри значения

```scala
given [A]: Encoder[Tree[A]] with
  type To = List[A]
  def encode(from: Tree[A]): To = toList(from)

given AsString[A](using Show[A]): Encoder[Tree[A]] with
  type To = String
  def encode(from: Tree[A]): To = toList(from).show
```

```haskell
Encoder (Tree a) where
  To = List a
  encode = toList
```

Тип зависит от типа внутри значения

```
given [A]: Encoder[Tree[A]] with
  type To = List[A]
  def encode(from: Tree[A]): To = toList(from)

given AsString[A](using Show[A]): Encoder[Tree[A]] with
  type To = String
  def encode(from: Tree[A]): To = toList(from).show
```

```
Encoder (Tree a) where
  To = List a
  encode = toList

[AsString] Show a ⇒ Encoder (Tree a) where
  To = String
  encode = show . toList
```

Тип зависит от типа внутри значения

```
def aTree: Tree[Int] = Node(5,
  NonEmptyList(Leaf(4),
  List(Leaf(6), Node(7, NonEmptyList(Leaf(8), List())))))
```

Тип зависит от типа внутри значения

```
def aTree: Tree[Int] = Node(5,
  NonEmptyList(Leaf(4),
  List(Leaf(6), Node(7, NonEmptyList(Leaf(8), List())))))
```

```
aTree : Tree Nat
aTree = Node 5 $ Leaf 4:::Leaf 6 :: Node 7 (Leaf 8:::[]) :: []
```

Тип зависит от типа внутри значения

```
def aTree: Tree[Int] = Node(5,
  NonEmptyList(Leaf(4),
  List(Leaf(6), Node(7, NonEmptyList(Leaf(8), List())))))

def aList: List[Int] = accEncoded(aTree, List(0, 1))
```

```
aTree : Tree Nat
aTree = Node 5 $ Leaf 4:::Leaf 6 :: Node 7 (Leaf 8:::[]) :: []
```

Тип зависит от типа внутри значения

```
def aTree: Tree[Int] = Node(5,
  NonEmptyList(Leaf(4),
  List(Leaf(6), Node(7, NonEmptyList(Leaf(8), List())))))

def aList: List[Int] = accEncoded(aTree, List(0, 1))
```

```
aTree : Tree Nat
aTree = Node 5 $ Leaf 4:::Leaf 6 :: Node 7 (Leaf 8:::[]) :: []

aList : List Nat
aList = accEncoded aTree [0, 1]
```

Тип зависит от типа внутри значения

```
def aTree: Tree[Int] = Node(5,
  NonEmptyList(Leaf(4),
  List(Leaf(6), Node(7, NonEmptyList(Leaf(8), List())))))

def aList: List[Int] = accEncoded(aTree, List(0, 1))

def aStr: String = accEncoded(using AsString[Int])(aTree, "01")
```

```
aTree : Tree Nat
aTree = Node 5 $ Leaf 4:::Leaf 6 :: Node 7 (Leaf 8:::[]) :: []

aList : List Nat
aList = accEncoded aTree [0, 1]
```

Тип зависит от типа внутри значения

```
def aTree: Tree[Int] = Node(5,
  NonEmptyList(Leaf(4),
  List(Leaf(6), Node(7, NonEmptyList(Leaf(8), List())))))

def aList: List[Int] = accEncoded(aTree, List(0, 1))

def aStr: String = accEncoded(using AsString[Int])(aTree, "01")
```

```
aTree : Tree Nat
aTree = Node 5 $ Leaf 4:::Leaf 6 :: Node 7 (Leaf 8:::[]) :: []

aList : List Nat
aList = accEncoded aTree [0, 1]

aStr : String
aStr = accEncoded @{AsString} aTree "01"
```

Вы уже давно это знаете (наверное)

Вы уже давно это знаете (наверное)

Для любого натурального n в пространстве векторов размерности n если $p(x, y)$ — метрика, над ней выполнено неравенство треугольника

Вы уже давно это знаете (наверное)

Для любого натурального n в пространстве векторов размерности n если $p(x, y)$ — метрика, над ней выполнено неравенство треугольника

$$\forall n : \mathbb{N} \cdot$$

Вы уже давно это знаете (наверное)

Для любого натурального n в пространстве векторов размерности n если $p(x, y)$ — метрика, над ней выполнено неравенство треугольника

$$\forall n \colon \mathbb{N} \cdot \forall p \colon \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \cdot$$

Вы уже давно это знаете (наверное)

Для любого натурального n в пространстве векторов размерности n если $p(x, y)$ — метрика, над ней выполнено неравенство треугольника

$$\forall n \colon \mathbb{N} \cdot \forall p \colon \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \cdot\ p(\cdot, \cdot) \text{ — метрика} \Longrightarrow$$

Вы уже давно это знаете (наверное)

Для любого натурального n в пространстве векторов размерности n если $p(x, y)$ — метрика, над ней выполнено неравенство треугольника

$$\forall n \colon \mathbb{N} \cdot \forall p \colon \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \cdot p(\cdot, \cdot) \text{ — метрика} \implies \forall x, y, z \colon \mathbb{R}^n \cdot p(x, y) + p(y, z) \geq p(x, z)$$

Вы уже давно это знаете (наверное)

Для любого натурального n в пространстве векторов размерности n если $p(x, y)$ — метрика, над ней выполнено неравенство треугольника

$$\forall n \colon \mathbb{N} \cdot \forall p \colon \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \cdot p(\cdot, \cdot) \text{ — метрика} \implies \forall x, y, z \colon \mathbb{R}^n \cdot p(x, y) + p(y, z) \geq p(x, z)$$

Вектор размерности n?

Тип зависит от значения

Для любого натурального n в пространстве векторов размерности n если $p(x, y)$ — метрика, над ней выполнено неравенство треугольника

$$\forall n \colon \mathbb{N} \cdot \forall p \colon \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \cdot p(\cdot, \cdot) \text{ — метрика} \Longrightarrow \forall x, y, z \colon \mathbb{R}^n \cdot p(x, y) + p(y, z) \geq p(x, z)$$

Вектор размерности n?

Вектор размерности n

Вектор размерности n

Хотим:

- создавать корректно

Вектор размерности n

Хотим:

- создавать корректно
- паттерн-матчить уверенно

Вектор размерности n

Хотим:

- создавать корректно
- паттерн-матчить уверенно
- сохранение при преобразованиях (напр. `map`, `traverse`)

Вектор размерности ?

```
enum List[+A]:
  case Nil
  case Cons(head: A, tail: List[A])
```

Вектор размерности ?

```
enum List[+A]:
  case Nil                                  extends List[Nothing]
  case Cons(head: A, tail: List[A])         extends List[A]
```

Вектор размерности n

```
enum Vect[N <: ???, +A]:
  case Nil                                  extends Vect[?, Nothing]
  case Cons(head: A, tail: Vect[N, A]) extends Vect[ ??? , A]
```

Вектор размерности n

```
sealed trait Nat
final abstract class Z           extends Nat
final abstract class S[N <: Nat] extends Nat

enum Vect[N <: ???, +A]:
  case Nil                                extends Vect[?, Nothing]
  case Cons(head: A, tail: Vect[N, A]) extends Vect[ ??? , A]
```

Вектор размерности n

```
sealed trait Nat
final abstract class Z           extends Nat
final abstract class S[N <: Nat] extends Nat

enum Vect[N <: Nat, +A]:
  case Nil                                extends Vect[Z, Nothing]
  case Cons(head: A, tail: Vect[N, A]) extends Vect[S[N], A]
```

Вектор размерности n

```
// Int
// 0 <: Int
// compiletime.ops.int.S

enum Vect[N <: Int, +A]:
  case Nil                                  extends Vect[0, Nothing]
  case Cons(head: A, tail: Vect[N, A]) extends Vect[S[N], A]
```

Вектор размерности n

```
sealed trait Nat
final abstract class Z           extends Nat
final abstract class S[N <: Nat] extends Nat

enum Vect[N <: Nat, +A]:
  case Nil                                extends Vect[Z, Nothing]
  case Cons(head: A, tail: Vect[N, A]) extends Vect[S[N], A]
```

Вектор размерности n

```
sealed trait Nat
final abstract class Z           extends Nat
final abstract class S[N <: Nat] extends Nat

enum Vect[N <: Nat, +A]:
  case Nil                                extends Vect[Z, Nothing]
  case Cons(head: A, tail: Vect[N, A]) extends Vect[S[N], A]


extension [A](hd: A)
  def ::[N <: Nat](tl: Vect[N, A]): Vect[S[N], A] = Cons(hd, tl)

object `::`:
  def unapply[N <: Nat, A](v: Vect[S[N], A]): (A, Vect[N, A]) =
    v match { case Cons(x, xs) ⇒ (x, xs) }
```

Вектор размерности n

Хотим: создавать корректно

Вектор размерности n

```
type Four = S[S[S[S[Z]]]]
type Five = S[Four]
```

Вектор размерности n

```
type Four = S[S[S[S[Z]]]]
type Five = S[Four]


def aVect5 : Vect[Five, Int] = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
def aVect4 : Vect[Four, Int] = 1 :: 2 :: 3 :: 4 :: Nil
```

Вектор размерности n

```
type Four = S[S[S[S[Z]]]]
type Five = S[Four]


def aVect5 : Vect[Five, Int] = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
def aVect4 : Vect[Four, Int] = 1 :: 2 :: 3 :: 4 :: Nil


def badVect5 : Vect[Five, Int] = 1 :: 2 :: 3 :: 4 :: Nil
```

Вектор размерности n

```
type Four = S[S[S[S[Z]]]]
type Five = S[Four]


def aVect5 : Vect[Five, Int] = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
def aVect4 : Vect[Four, Int] = 1 :: 2 :: 3 :: 4 :: Nil


def badVect5 : Vect[Five, Int] = 1 :: 2 :: 3 :: 4 :: Nil
```

```
Found:    Vect[S[S[S[S[Z]]]], Int]
Required: Vect[Five, Int]
```

Вектор размерности n

Хотим: паттерн-матчить уверенно

Вектор размерности n

```
//                    def aVect5 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

Вектор размерности n

```
//                    def aVect5 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

```
def secFif: (Int, Int) = aVect5 match
  case _ :: sec :: _ :: _ :: fif :: Nil ⇒ (sec, fif)
```

Вектор размерности n

```
//                    def aVect5 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

```
def secFif: (Int, Int) = aVect5 match
  case _ :: sec :: _ :: _ :: fif :: Nil ⇒ (sec, fif)
  //   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  //                          |
  //                  exhaustive pattern matching
```

Вектор размерности n

Хотим: сохранение при преобразованиях (напр. `map`, `traverse`)

Вектор размерности n

Хотим: сохранение при преобразованиях (напр. `map`, `traverse`)

```
given [N <: Nat]: Functor[[A] =>> Vect[N, A]] with
  def map[A, B](v: Vect[N, A])(f: A => B): Vect[N, B] = v match
    case Nil         => Nil
    case Cons(x, xs) => Cons(f(x), xs.map(f))
```

Вектор размерности n

Хотим: сохранение при преобразованиях (напр. `map`, `traverse`)

```
given [N <: Nat]: Functor[[A] =>> Vect[N, A]] with
  def map[A, B](v: Vect[N, A])(f: A => B): Vect[N, B] = v match
    case Nil         => Nil
    case Cons(x, xs) => Cons(f(x), xs.map(f))
```

```
given [N <: Nat]: Traverse[[A] =>> Vect[N, A]] with
  ...
  def traverse[G[_]: Applicative, A, B]
              (fa: Vect[N, A])(f: A => G[B]): G[Vect[N, B]] =
    fa match
      case Nil         => Nil.pure
      case Cons(x, xs) => (f(x), xs.traverse(f)).mapN(Cons.apply)
```

Вектор размерности n

Хотим: сохранение при преобразованиях (напр. `map`, `traverse`)

Вектор размерности n

```
//                    def aVect5 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

Вектор размерности n

```
//                    def aVect5 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

```
def secFifStr: (String, String) = aVect5.map(_.show) match
  case mapped ⇒ ???
```

Вектор размерности n

```
//                    def aVect5 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

```
def secFifStr: (String, String) = aVect5.map(_.show) match
  case _ :: sec :: _ :: _ :: fif :: Nil ⇒ (sec, fif)
```

Вектор размерности n

```
//                    def aVect5 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

```
def secFifStr: (String, String) = aVect5.map(_.show) match
  case _ :: sec :: _ :: _ :: fif :: Nil ⇒ (sec, fif)
```

```
def secFifA[F[_]: Applicative, B](f: Int ⇒ F[B]): F[(B, B)] =
  aVect5.traverse(f).map:
    case _ :: sec :: _ :: _ :: fif :: Nil ⇒ (sec, fif)
```

Вектор размерности n

```
def concat[A]
          (l: List[A], r: List[A]): List[A] =
  l match
    case Nil         ⇒ r
    case x :: xs     ⇒ x :: concat(xs, r)
```

Вектор размерности n

```
def concat[N <: Nat, M <: Nat, A]
          (l: Vect[N, A], r: Vect[M, A]): Vect[ ??? , A] =
  l match
    case Nil         ⇒ r
    case Cons(x, xs) ⇒ x :: concat(xs, r)
```

Вектор размерности n

```
type +[N <: Nat, M <: Nat] <: Nat = N match
  case Z    ⇒ M
  case S[k] ⇒ S[k + M]
```

```
def concat[N <: Nat, M <: Nat, A]
          (l: Vect[N, A], r: Vect[M, A]): Vect[ ??? , A] =
  l match
    case Nil         ⇒ r
    case Cons(x, xs) ⇒ x :: concat(xs, r)
```

Вектор размерности n

```
type +[N <: Nat, M <: Nat] <: Nat = N match
  case Z    ⇒ M
  case S[k] ⇒ S[k + M]
```

```
def concat[N <: Nat, M <: Nat, A]
          (l: Vect[N, A], r: Vect[M, A]): Vect[N + M, A] =
  l match
    case Nil         ⇒ r
    case Cons(x, xs) ⇒ x :: concat(xs, r)
```

Вектор размерности n

```
type +[N <: Nat, M <: Nat] <: Nat = N match
  case Z    ⇒ M
  case S[k] ⇒ S[k + M]
```

```
def concat[N <: Nat, M <: Nat, A]
          (l: Vect[N, A], r: Vect[M, A]): Vect[N + M, A] =
  l match
    case Nil         ⇒ r
    case Cons(x, xs) ⇒ x :: concat(xs, r)
```

NB: нет зависимости от значения

Вектор размерности n

```
type +[N <: Nat, M <: Nat] <: Nat = N match
  case Z    ⇒ M
  case S[k] ⇒ S[k + M]
```

```
extension [N <: Nat, A](l: Vect[N, A])
  def ++[M <: Nat](r: Vect[M, A]): Vect[N + M, A] = l match
    case Nil         ⇒ r
    case Cons(x, xs) ⇒ x :: (xs ++ r)
```

NB: нет зависимости от значения

Вектор размерности n

```
//                  def last[A](xs: List[A]): Option[A] = ???
--                  last : {0 a : Type} -> List a -> Maybe a
```

Вектор размерности n

```
//                  def last[A](xs: List[A]): Option[A] = ???
--                  last : {0 a : Type} -> List a -> Maybe a

// sealed trait Nat

enum Vect[N <: Nat, +A]:
  case Nil                                  extends Vect[Z, Nothing]
  case Cons(head: A, tail: Vect[N, A]) extends Vect[S[N], A]
```

Вектор размерности n

```
//                  def last[A](xs: List[A]): Option[A] = ???
--                  last : {0 a : Type} -> List a -> Maybe a

// sealed trait Nat

enum Vect[N <: Nat, +A]:
  case Nil                                  extends Vect[Z, Nothing]
  case Cons(head: A, tail: Vect[N, A]) extends Vect[S[N], A]

-- data Nat = Z | S n

data Vect : Nat -> Type -> Type where
```

Вектор размерности n

```
//                    def last[A](xs: List[A]): Option[A] = ???
--                    last : {0 a : Type} -> List a -> Maybe a
```

```
// sealed trait Nat

enum Vect[N <: Nat, +A]:
  case Nil                                  extends Vect[Z, Nothing]
  case Cons(head: A, tail: Vect[N, A]) extends Vect[S[N], A]
```

```
-- data Nat = Z | S n

data Vect : Nat -> Type -> Type where
  Nil  : Vect Z a
  (::) : a -> Vect n a -> Vect (S n) a
```

Вектор размерности n

```
// type +[N <: Nat, M <: Nat] <: Nat = N match
//   case Z    ⇒ M
//   case S[k] ⇒ S[k + M]

def concat[N <: Nat, M <: Nat, A]
          (l: Vect[N, A], r: Vect[M, A]): Vect[N + M, A] =
  l match { case Nil         ⇒ r
            case Cons(x, xs) ⇒ x :: concat(xs, r) }
```

Вектор размерности n

```
// type +[N <: Nat, M <: Nat] <: Nat = N match
//   case Z    ⇒ M
//   case S[k] ⇒ S[k + M]

def concat[N <: Nat, M <: Nat, A]
          (l: Vect[N, A], r: Vect[M, A]): Vect[N + M, A] =
  l match { case Nil         ⇒ r
            case Cons(x, xs) ⇒ x :: concat(xs, r) }
```

```
-- (+) : Nat → Nat → Nat
-- Z   + m = m
-- S n + m = S $ n + m

(++) : Vect n a → Vect m a → Vect (n + m) a
[]      ++ ys = ys
(x::xs) ++ ys = x :: (xs ++ ys)
```

Вектор размерности n: level up

Вектор размерности n: level up

```
replicate : (n : Nat) -> a -> Vect n a
replicate Z     _ = []
replicate (S n) x = x :: replicate n x
```

Вектор размерности n: level up

```
def replicate[N <: Nat, A](a: A): Vect[N, A]
```

```
replicate : (n : Nat) -> a -> Vect n a
replicate Z     _ = []
replicate (S n) x = x :: replicate n x
```

Вектор размерности n: level up

```
def replicate[N <: Nat, A](n: N, a: A): Vect[N, A]
```

```
replicate : (n : Nat) -> a -> Vect n a
replicate Z     _ = []
replicate (S n) x = x :: replicate n x
```

Вектор размерности n: level up

```
// sealed trait Nat
// final abstract class Z           extends Nat
// final abstract class S[N <: Nat] extends Nat


def replicate[N <: Nat, A](n: N, a: A): Vect[N, A]
```

```
replicate : (n : Nat) -> a -> Vect n a
replicate Z     _ = []
replicate (S n) x = x :: replicate n x
```

Вектор размерности n: level up

```
def replicate[N <: Nat, A](n: ???, a: A): Vect[N, A]
```

```
replicate : (n : Nat) -> a -> Vect n a
replicate Z     _ = []
replicate (S n) x = x :: replicate n x
```

Вектор размерности n: level up

```
enum NatVal[N <: Nat]:
  case ZV                extends NatVal[Z]
  case SV(n: NatVal[N]) extends NatVal[S[N]]


def replicate[N <: Nat, A](n: ???, a: A): Vect[N, A]
```

```
replicate : (n : Nat) -> a -> Vect n a
replicate Z     _ = []
replicate (S n) x = x :: replicate n x
```

Вектор размерности n: level up

```
enum NatVal[N <: Nat]:
  case ZV                extends NatVal[Z]
  case SV(n: NatVal[N]) extends NatVal[S[N]]


def replicate[N <: Nat, A](n: NatVal[N], a: A): Vect[N, A]
```

```
replicate : (n : Nat) -> a -> Vect n a
replicate Z     _ = []
replicate (S n) x = x :: replicate n x
```

Вектор размерности n: level up

```
enum NatVal[N <: Nat]:
  case ZV                extends NatVal[Z]
  case SV(n: NatVal[N]) extends NatVal[S[N]]


def replicate[N <: Nat, A](n: NatVal[N], a: A): Vect[N, A] =
  n match { case ZV    ⇒ Nil
            case SV(m) ⇒ a :: replicate(m, a) }
```

```
replicate : (n : Nat) -> a -> Vect n a
replicate Z     _ = []
replicate (S n) x = x :: replicate n x
```

Вектор размерности n: level up

```
def fourVal: NatVal[Four] = SV(SV(SV(SV(ZV))))
```

Вектор размерности n: level up

```
def fourVal: NatVal[Four] = SV(SV(SV(SV(ZV))))
def aRep4: Vect[Four, Int] = replicate(fourVal, 9)
```

Вектор размерности n: level up

```
def fourVal: NatVal[Four] = SV(SV(SV(SV(ZV))))
def aRep4: Vect[Four, Int] = replicate(fourVal, 9)
def a9 = aVect5 ++ aRep4
```

Вектор размерности n: level up

```
def fourVal: NatVal[Four] = SV(SV(SV(SV(ZV))))
def aRep4: Vect[Four, Int] = replicate(fourVal, 9)
def a9 = aVect5 ++ aRep4
```

```
aRep4 : Vect 4 Int
aRep4 = replicate 4 9
```

Вектор размерности n: level up

```
def fourVal: NatVal[Four] = SV(SV(SV(SV(ZV))))
def aRep4: Vect[Four, Int] = replicate(fourVal, 9)
def a9 = aVect5 ++ aRep4
```

```
aRep4 : Vect 4 Int
aRep4 = replicate 4 9
a9 : ?
a9 = aVect5 ++ aRep4
```

Вектор размерности n: level up

```
--                          parsePositive ~~ String -> Maybe Nat
--                          getLine       ~~ IO String


manageS : String -> IO Integer

useVects : String -> IO String
useVects str = do
  let Just n = parsePositive str
    | Nothing => pure "not a number"
  let xs = replicate n "x"
  ys <- traverse manageS xs
  pure $ show $ xs `zip` ys
```

Вектор размерности n: level up

```
--                                  parsePositive ~~ String -> Maybe Nat
```

Вектор размерности n: level up

```
--                          parsePositive ~~ String -> Maybe Nat
```

```
def parsePositive[N <: Nat](str: String): Option[NatVal[N]] = ???
```

Вектор размерности n: level up

```
--                          parsePositive ~~ String -> Maybe Nat
```

```
def parsePositive(str: String): Option[???] = ???
```

Вектор размерности n: level up

```
--                          parsePositive ~~ String -> Maybe Nat

trait SomeNatVal:
  type N <: Nat
  val value: NatVal[N]

def parsePositive(str: String): Option[???] = ???
```

Вектор размерности n: level up

```
--                          parsePositive ~~ String -> Maybe Nat

trait SomeNatVal:
  type N <: Nat
  val value: NatVal[N]

def parsePositive(str: String): Option[SomeNatVal] = ???
```

Вектор размерности n: level up

```
--                              parsePositive ~~ String -> Maybe Nat

trait SomeNatVal:
  type N <: Nat
  val value: NatVal[N]

def parsePositive(str: String): Option[SomeNatVal] = ???

def manageS(str: String): IO[Int] = ???
def useVects(str: String): IO[String] = parsePositive(str) match
  case None    => "not a number".pure
  case Some(n) =>
    val xs = replicate(n.value, "x")
    xs.traverse(manageS).map(ys => zip(xs, ys).show)
```

Минуточку!

Минуточку!

```
trait SomeNatVal:
  type N <: Nat
  val value: NatVal[N]
```

Минуточку!

```
trait SomeNatVal:
  type N <: Nat
  val value: NatVal[N]


def replicate[N <: Nat, A](n: NatVal[N], a: A): Vect[N, A] =
  n match
    case ZV    ⇒ Nil
    case SV(m) ⇒ a :: replicate(m, a)
```

Минуточку!

```
trait SomeNatVal:
  type N <: Nat
  val value: NatVal[N]


def replicate[A](n: SomeNatVal, a: A): Vect[n.N, A] =
  n.value match
    case ZV    ⇒ Nil
    case SV(m) ⇒ a :: replicate(SomeNatVal(m), a)
```

Минуточку!

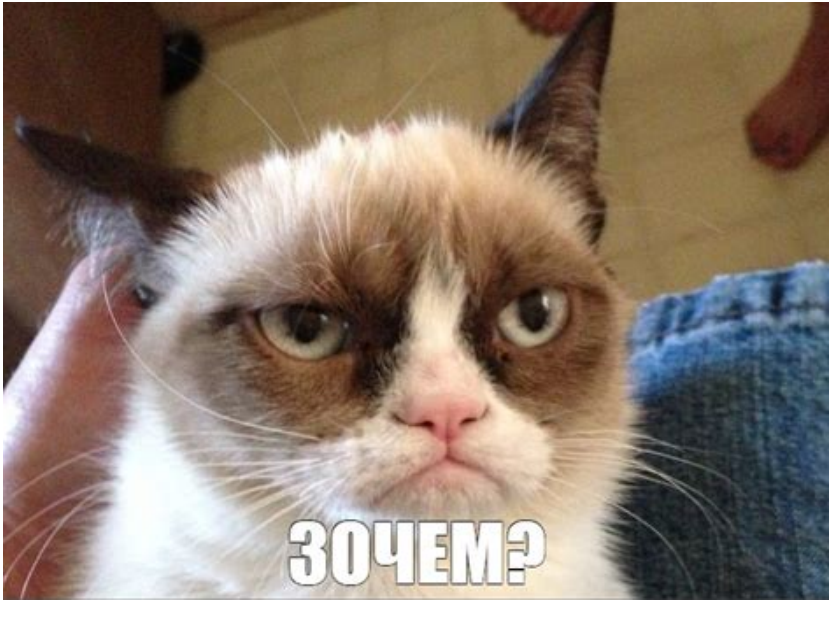
```
trait SomeNatVal:
  type N <: Nat
  val value: NatVal[N]


def replicate[A](n: SomeNatVal, a: A): Vect[n.N, A] =
  n.value match
    case ZV    ⇒ Nil
    case SV(m) ⇒ a :: replicate(SomeNatVal(m), a)
```

```
Found:    (Vect.Nil : Vect[Z, Nothing])
Required: Vect[n.N, A]
```

Экзистенциальный кризис

Экзистенциальный кризис



Как вы индексируете вектора?

Как вы индексируете вектора?

```
trait Seq[+A] extends ..., PartialFunction[Int, A], ...
```

Как вы индексируете вектора?

```
trait Seq[+A] extends ..., PartialFunction[Int, A], ...
def lift: Int ⇒ Option[A]
```

Как вы индексируете вектора размерности n?

```
trait Seq[+A] extends ..., PartialFunction[Int, A], ...
def lift: Int ⇒ Option[A]
```

```
total
index : Nat -> Vect n a -> Maybe a
```

Как вы индексируете вектора размерности n?

```
trait Seq[+A] extends ..., PartialFunction[Int, A], ...
def lift: Int ⇒ Option[A]
```

```
total
index : ? -> Vect n a -> a
```

Как вы индексируете вектора размерности n?

```
trait Seq[+A] extends ..., PartialFunction[Int, A], ...
def lift: Int ⇒ Option[A]
```

```
data Fin : Nat → Type where
  FZ : Fin (S n)
  FS : Fin n → Fin (S n)

total
index : ? → Vect n a → a
```

Как вы индексируете вектора размерности n?

```
trait Seq[+A] extends ..., PartialFunction[Int, A], ...
def lift: Int ⇒ Option[A]
```

```
data Fin : Nat → Type where
  FZ : Fin (S n)
  FS : Fin n → Fin (S n)

total
index : Fin n → Vect n a → a
```

Вершина айсберга

Вершина айсберга

```
data BinTree : Type → Type where
  Empty : BinTree a
  Node  : (x : a) → (left, right : BinTree a) → BinTree a
```

Вершина айсберга

```
data SortedBinTree : Type → Type where
  Empty : SortedBinTree a
  Node  : Ord a ⇒ (x : a) → (left, right : SortedBinTree a) →
          ?left_sorted   ⇒ ?right_sorted   ⇒ SortedBinTree a
```

Вершина айсберга

```
data SortedBinTree : Type → Type
data All : (a → Bool) → SortedBinTree a → Type
```

```
data SortedBinTree : Type → Type where
  Empty : SortedBinTree a
  Node  : Ord a ⇒ (x : a) → (left, right : SortedBinTree a) →
          ?left_sorted   ⇒ ?right_sorted   ⇒ SortedBinTree a
```

Вершина айсберга

```
data SortedBinTree : Type → Type
data All : (a → Bool) → SortedBinTree a → Type

data SortedBinTree : Type → Type where
  Empty : SortedBinTree a
  Node  : Ord a ⇒ (x : a) → (left, right : SortedBinTree a) →
          All (< x) left ⇒ All (x <) right ⇒ SortedBinTree a
```

Вершина айсберга

```
data SortedBinTree : Type → Type
data All : (a → Bool) → SortedBinTree a → Type
```

```
data SortedBinTree : Type → Type where
  Empty : SortedBinTree a
  Node  : Ord a ⇒ (x : a) → (left, right : SortedBinTree a) →
          All (< x) left ⇒ All (x <) right ⇒ SortedBinTree a
```

```
data All : (a → Bool) → SortedBinTree a → Type where
  Empty' : All prop Empty
  Node'  : (o : Ord a) ⇒ {0 prop : a → Bool} →
           {0 pl : All (< x) l} → {0 pr : All (x <) r} →
           So (prop x) → All prop l → All prop r →
           All prop $ Node x l r @{o} @{pl} @{pr}
```

Вершина айсберга

```
Leaf : Ord a ⇒ a → SortedBinTree a
Leaf x = Node x Empty Empty
```

Вершина айсберга

```
Leaf : Ord a ⇒ a → SortedBinTree a
Leaf x = Node x Empty Empty



good : SortedBinTree Int
good = Node 4 (Node 2 (Leaf 1) (Leaf 3))
              (Leaf 5)
```

Вершина айсберга

```
Leaf : Ord a ⇒ a → SortedBinTree a
Leaf x = Node x Empty Empty



good : SortedBinTree Int
good = Node 4 (Node 2 (Leaf 1) (Leaf 3))
              (Leaf 5)


failing "Can't find an implementation for
         All (\\arg ⇒ 5 < arg) (Leaf 4)"
  bad : SortedBinTree Int
  bad = Node 5 (Node 2 (Leaf 1) (Leaf 3))
               (Leaf 4)
```

Если стало интересно

Literate Idris

Idris 2 language

Код на Scala

Эта презентация

Спасибо!

Literate Idris

Idris 2 language

Код на Scala

Эта презентация

Вопросы?