Зависимые типы и соответствие Карри-Ховарда

... или интерпретация Брауэра-Гейтинга-Колмогорова

Свежие идеи, которым 90 лет

Денис Буздалов

19 февраля 2020
(22 января 2020)

Disclaimer

- этот доклад изначально
  - готовился для конкретных слушателей, знакомых с Haskell
  - ставил целью знакомство с зависимыми типами, а не конкретно с Idris
- доклад выполнен в своеобразном стиле
- докладчик не очень знаком с уровнем подготовки и знаниями нынешней аудитории
- доклад ознакомительный и не претендует на абсолютную точность и полноту

Раздел 1

Вступление

Предположения о слушателях

- Максимально ленивы

Предположения о слушателях

- Хотят, чтобы максимум работы делал за них компилятор

Предположения о слушателях

- Хотят, чтобы максимум работы делал за них компилятор
- Готовы *работать* для этого ;-)

Предположения о слушателях

- Хотят, чтобы максимум работы делал за них компилятор
- Готовы *работать* для этого ;-)
- Готовы воспринимать *такое* новое, что оно изменит всю дальнейшую жизнь

Предположения о слушателях

- Хотят, чтобы максимум работы делал за них компилятор
- Готовы *работать* для этого ;-)
- Готовы воспринимать *такое* новое, что оно изменит всю дальнейшую жизнь (может быть, не сразу ;-))

Предположения о слушателях

- Хотят, чтобы максимум работы делал за них компилятор
- Готовы *работать* для этого ;-)
- Готовы воспринимать *такое* новое, что оно изменит всю дальнейшую жизнь (может быть, не сразу ;-))
- Готовы познакомиться с новым языком программирования

Предположения о слушателях

- Хотят, чтобы максимум работы делал за них компилятор
- Готовы *работать* для этого ;-)
- Готовы воспринимать *такое* новое, что оно изменит всю дальнейшую жизнь (может быть, не сразу ;-))
- Готовы познакомиться с новым языком программирования
  - Не самоцель, но важно для понимания идей

Предположения о слушателях

- Хотят, чтобы максимум работы делал за них компилятор
- Готовы *работать* для этого ;-)
- Готовы воспринимать *такое* новое, что оно изменит всю дальнейшую жизнь (может быть, не сразу ;-))
- Готовы познакомиться с новым языком программирования
  - Не самоцель, но важно для понимания идей
- Готовы к куче кода

Предположения о слушателях

- Хотят, чтобы максимум работы делал за них компилятор
- Готовы *работать* для этого ;-)
- Готовы воспринимать *такое* новое, что оно изменит всю дальнейшую жизнь (может быть, не сразу ;-))
- Готовы познакомиться с новым языком программирования
  - Не самоцель, но важно для понимания идей
- Готовы к куче кода
- Готовы к лукавству высших порядков ;-)

На повестке дня

- Тайпклассы с законами без зависимых типов
  - Описание (логика первого порядка)
  - Property-based тестирование
  - Singletons magic
- Соответствие Карри-Ховарда
  - Интуиционистская (конструктивная) логика
  - Собственно соответствие
  - Зависимые типы
- Некоторые применения зависимых типов
  - Индуктивные предикаты как GADT с зависимыми типами
  - Утверждения-ограничения в функциях
  - Структурно ограниченные типы
  - Состояние в типах
- О типизированных дырках

Раздел 2

О синтаксисе

Синтаксис Haskell: основы

- Базовые типы

`Int` `Int → String` `[Int]` `Maybe Int`

Синтаксис Haskell: основы

- Базовые типы

  `Int` `Int → String` `[Int]` `Maybe Int`

- Сигнатуры функций

  `f :: Int → Int` `g :: Int → Int → Int`

Синтаксис Haskell: основы

- Базовые типы

  `Int` `Int → String` `[Int]` `Maybe Int`

- Сигнатуры функций

  `f :: Int → Int` `g :: Int → Int → Int`

- Применение функций

  `f x` `g 5 6` `f . g 5`

Синтаксис Haskell: основы

- Базовые типы

```
Int        Int → String        [Int]        Maybe Int
```

- Сигнатуры функций

```
f :: Int → Int               g :: Int → Int → Int
```

- Применение функций

```
f x            g 5 6            f . g 5
g 5 (f x)                 g 5 $ f x
```

Синтаксис Haskell: основы

- Базовые типы

```
Int        Int → String        [Int]        Maybe Int
```

- Сигнатуры функций

```
f :: Int → Int        g :: Int → Int → Int
```

- Применение функций

```
f x            g 5 6            f . g 5

g 5 (f x)                g 5 $ f x

x `elem` xs
```

Синтаксис Haskell: основы

- Базовые типы

  `Int` `Int → String` `[Int]` `Maybe Int`

- Сигнатуры функций

  `f :: Int → Int` `g :: Int → Int → Int`

- Применение функций

  `f x` `g 5 6` `f . g 5`

  `g 5 (f x)` `g 5 $ f x`

  ``x `elem` xs``

- Синтаксис списков

  `x:xs` `[a, b, c]`

Синтаксис Haskell: данные

- Типы-произведения

```haskell
data Product a b = Product a b
```

Синтаксис Haskell: данные

- Типы-произведения

```
data Product a b = Product a b                  (a, b)
```

Синтаксис Haskell: данные

- Типы-произведения

```haskell
data Product a b = Product a b                  (a, b)
```

- Типы-суммы

```haskell
data Coproduct a b = A a | B b
```

Синтаксис Haskell: данные

- Типы-произведения

```
data Product a b = Product a b                (a, b)
```

- Типы-суммы

```
data Coproduct a b = A a | B b              Either a b
```

Синтаксис Haskell: данные

- Типы-произведения

```
data Product a b = Product a b                    (a, b)
```

- Типы-суммы

```
data Coproduct a b = A a | B b                    Either a b
```

- Algebraic data types

```
data X = A | B Int | C Int String
```

Синтаксис Haskell: данные

- Типы-произведения

```haskell
data Product a b = Product a b                (a, b)
```

- Типы-суммы

```haskell
data Coproduct a b = A a | B b              Either a b
```

- Algebraic data types

```haskell
data X = A | B Int | C Int String
```

- Pattern matching

```haskell
f :: X → String
f A       = "A"
f (B x)   = "B " ++ show x
f (C _ s) = "C " ++ s
```

Синтаксис Haskell: полиморфизм

- Полиморфные функции

```
f :: x → [x] → [x]
```

Синтаксис Haskell: полиморфизм

- Полиморфные функции

```haskell
f :: x → [x] → [x]
f :: forall x. x → [x] → [x]
```

Синтаксис Haskell: полиморфизм

- Полиморфные функции

```haskell
f :: x → [x] → [x]
```

```haskell
f :: forall x. x → [x] → [x]
```

- Ограниченный полиморфизм

```haskell
class Eq a where
  (==) :: a → a → Bool
```

Синтаксис Haskell: полиморфизм

- Полиморфные функции

```
f :: x → [x] → [x]
```

```
f :: forall x. x → [x] → [x]
```

- Ограниченный полиморфизм

```
class Eq a where
  (==) :: a → a → Bool
```

```
allEq :: Eq a ⇒ [a] → Bool
```

Синтаксис Haskell: полиморфизм

- Полиморфные функции

```haskell
f :: x → [x] → [x]
```

```haskell
f :: forall x. x → [x] → [x]
```

- Ограниченный полиморфизм

```haskell
class Eq a where
  (==) :: a → a → Bool
```

```haskell
allEq :: Eq a ⇒ [a] → Bool
allEq []     = True
allEq (x:xs) = all (== x) xs
```

Раздел 3

Пример проблемы

Тайпклассы с подразумеваемыми законами

```
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
```

Тайпклассы с подразумеваемыми законами

```
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
```

```
class Eq a ⇒ Ord a where
  (<)  :: a → a → Bool
  (<=) :: a → a → Bool
  (>)  :: a → a → Bool
  (>=) :: a → a → Bool
```

Тайпклассы с неформально записанными законами

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Тайпклассы с неформально записанными законами

```haskell
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
```

- Рефлексивность: $\forall x\colon a \cdot x = x$
- Симметричность: $\forall x, y\colon a \cdot x = y \iff y = x$
- Транзитивность: $\forall x, y, z\colon a \cdot x = y \wedge y = z \Rightarrow x = z$

Тайпклассы с неформально записанными законами

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

- Рефлексивность: $\forall x\colon a \cdot x = x$
- Симметричность: $\forall x, y\colon a \cdot x = y \iff y = x$
- Транзитивность: $\forall x, y, z\colon a \cdot x = y \land y = z \Rightarrow x = z$
- $\forall a, b \cdot a = b \iff \neg(a \neq b)$

Тайпклассы с неформально записанными законами

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

- Рефлексивность: $\forall x\colon a \cdot x = x$
- Симметричность: $\forall x, y\colon a \cdot x = y \iff y = x$
- Транзитивность: $\forall x, y, z\colon a \cdot x = y \wedge y = z \Rightarrow x = z$
- $\forall a, b \cdot a = b \iff \neg(a \neq b)$

```haskell
class Eq a => Ord a where
  (<) :: a -> a -> Bool
```

Тайпклассы с неформально записанными законами

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

- Рефлексивность: $\forall x : a \cdot x = x$
- Симметричность: $\forall x, y : a \cdot x = y \iff y = x$
- Транзитивность: $\forall x, y, z : a \cdot x = y \land y = z \Rightarrow x = z$

- $\forall a, b \cdot a = b \iff \neg(a \neq b)$

```haskell
class Eq a => Ord a where
  (<) :: a -> a -> Bool
```

- Антирефлексивность: $\forall x : a \cdot \neg(x < x)$
- Антисимметричность: $\forall x, y : a \cdot x < y \Rightarrow \neg(y < x)$
- Транзитивность: $\forall x, y, z : a \cdot x < y \land y < z \Rightarrow x < z$

Тайпклассы с неформально записанными законами

```haskell
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
```

- Рефлексивность: $\forall x\colon a \cdot x = x$
- Симметричность: $\forall x, y\colon a \cdot x = y \iff y = x$
- Транзитивность: $\forall x, y, z\colon a \cdot x = y \land y = z \Rightarrow x = z$

- $\forall a, b \cdot a = b \iff \neg(a \neq b)$

```haskell
class Eq a ⇒ Ord a where
  (<) :: a → a → Bool
```

- Антирефлексивность: $\forall x\colon a \cdot \neg(x < x)$
- Антисимметричность: $\forall x, y\colon a \cdot x < y \Rightarrow \neg(y < x)$
- Транзитивность: $\forall x, y, z\colon a \cdot x < y \land y < z \Rightarrow x < z$

- $\forall a, b \cdot a \leqslant b \iff a < b \lor a = b$
- $\forall a, b \cdot a > b \iff b < a$
- $\forall a, b \cdot a \geqslant b \iff a > b \lor a = b$

Property-based testing: шаг к формализации законов

```haskell
eqLaws :: forall a. (Eq a, Arbitrary a, Show a) => Proxy a -> Spec
eqLaws _ = describe "Eq typeclass" do

  describe "== operation" do
    prop "reflexivity"    $ \(x::a) ->
      x == x
    prop "symmetry"       $ \(x::a) (y::a) ->
      (x == y) === (y == x)
    modifyMaxDiscardRatio (*10^6) .
      prop "transitivity" $ \(x::a) (y::a) (z::a) ->
        x == y && y == z ==> x == z

  describe "/= operation" do
    prop "equals to not ==" $ \(x::a) (y::a) ->
      (x == y) === not (x /= y)
```

Property-based testing: шаг к формализации законов

```haskell
ordLaws :: forall a. (Ord a, Arbitrary a, Show a) ⇒ Proxy a → Spec
ordLaws _ = describe "Ord typeclass" do
  describe "< operation" do
    prop "anti-reflexivity" $ \(x :: a) →
      not (x < x)
    prop "anti-symmetry"    $ \(x :: a) (y :: a) →
      x < y ⟹ not (y < x)
    prop "transitivity"     $ \(x :: a) (y :: a) (z :: a) →
      x < y && y < z ⟹ x < z

  describe "≤ operation" do
    prop "is < or ≡" $ \(x :: a) (y :: a) → (x ≤ y) ≡ (x<y || x≡y)
  describe "> operation" do
    prop "is not <"   $ \(x :: a) (y :: a) → (x > y) ≡ (y < x)
  describe "≥ operation" do
    prop "is > or ="  $ \(x :: a) (y :: a) → (x ≥ y) ≡ (x>y || x≡y)
```

LiquidHaskell?

LiquidHaskell? Позволяет крутые штуки[1]

```haskell
{-@ type NonEmpty a = {v:[a] | 0 < len v} @-}

{-@ head :: NonEmpty a → a @-}
head (x:_) = x
```

[1] https://ucsd-progsys.github.io/liquidhaskell-blog/

LiquidHaskell? Позволяет крутые штуки[1]

```
{-@ type NonEmpty a = {v:[a] | 0 < len v} @-}

{-@ head :: NonEmpty a → a @-}
head (x:_) = x



{-@ merge :: Ord a ⇒ xs:[a] → ys:[a] → [a]/[len xs + len ys] @-}
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
  | x ⩽ y     = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

[1] https://ucsd-progsys.github.io/liquidhaskell-blog/

LiquidHaskell? Даже законы[2]

```
assocThm xs ys zs = (xs ++ ys) ++ zs == xs ++ (ys ++ zs)

{-@ assocPrf :: xs:_ -> ys:_ -> zs:_ -> { assocThm xs ys zs } @-}

assocPrf [] ys zs
  = ([] ++ ys) ++ zs
  ==. [] ++ (ys ++ zs)
    *** QED

assocPrf (x:xs) ys zs
  = ((x:xs) ++ ys) ++ zs
  ==. (x : (xs ++ ys)) ++ zs
  ==. (x : ((xs ++ ys) ++ zs))
  ==. (x : (xs ++ (ys ++ zs))) ? assocPrf xs ys zs
  ==. (x : xs) ++ (ys ++ zs)
    *** QED
```

[2] https://ucsd-progsys.github.io/liquidhaskell-blog/

LiquidHaskell? Даже законы[2]

```
assocThm xs ys zs = (xs ++ ys) ++ zs == xs ++ (ys ++ zs)

{-@ assocPrf :: xs:_ -> ys:_ -> zs:_ -> { assocThm xs ys zs } @-}

assocPrf [] ys zs
  = ([] ++ ys) ++ zs
  ==. [] ++ (ys ++ zs)
    *** QED

assocPrf (x:xs) ys zs
  = ((x:xs) ++ ys) ++ zs
  ==. (x : (xs ++ ys)) ++ zs
  ==. (x : ((xs ++ ys) ++ zs))
  ==. (x : (xs ++ (ys ++ zs))) ? assocPrf xs ys zs
  ==. (x : xs) ++ (ys ++ zs)
    *** QED
```

Но у меня не получилось для тайпклассов

[2] https://ucsd-progsys.github.io/liquidhaskell-blog/

Type-level-программирование и синглтоны

```haskell
{-# LANGUAGE DataKinds, KindSignatures, PolyKinds #-}
{-# LANGUAGE TypeOperators, TypeFamilies #-}

class Eq a where
  type (x :: a) == (y :: a) :: Bool

  (%==) :: Sing (x :: a) → Sing (y :: a) → Sing (x == y)

  eqRefl :: Sing (x :: a) → x == x :~: 'True
  eqSymm :: Sing (x :: a) → Sing (y :: a) → x == y :~: y == x
  eqTran :: Sing (x :: a) → Sing (y :: a) → Sing (z :: a)
       → x == y :~: 'True → y == z :~: 'True → x == z :~: 'True

(==) :: (SingKind m, Eq m) ⇒ Demote m → Demote m → Bool
x == y = withSomeSing x $ \sX →
           withSomeSing y $ \sY →
             fromSing (sX %== sY)
```

Type-level-программирование и синглтоны

```
data List a = Nil | Cons a (List a)
data instance Sing (xs :: List a) where
    SNil   :: Sing 'Nil
    SCons  :: Sing x → Sing xs → Sing ('Cons x xs)

instance Eq a ⇒ Eq (List a) where
  type 'Nil       == 'Nil       = 'True
  type 'Nil       == 'Cons _ _  = 'False
  type 'Cons _ _  == 'Nil       = 'False
  type 'Cons x xs == 'Cons y ys = x == y && xs == ys

  SNil       %== SNil       = STrue
  SNil       %== SCons _ _  = SFalse
  SCons _ _  %== SNil       = SFalse
  SCons x xs %== SCons y ys = x %== y %&& xs %== ys

  ...
```

Type-level-программирование и синглтоны

```
instance Eq a ⇒ Eq (List a) where
  ...

  eqRefl SNil          = Refl
  eqRefl (SCons x xs) = case (eqRefl x, eqRefl xs) of
    (Refl, Refl) → Refl

  eqSymm SNil         SNil         = Refl
  eqSymm SNil         (SCons _ _) = Refl
  eqSymm (SCons _ _) SNil         = Refl
  eqSymm (SCons x l) (SCons y r) = case (eqSymm x y, eqSymm l r) of
    (Refl, Refl) → Refl

  eqTran SNil SNil SNil Refl Refl = Refl
  ...
```

Type-level-программирование и синглтоны

```haskell
$(singletons [d|
  data List a = Nil | Cons a (List a)

  equals :: Eq a => List a -> List a -> Bool
  equals Nil         Nil         = True
  equals (Cons x xs) (Cons y ys) = x == y && equals xs ys
  equals _           _           = False |])

instance Eq a => Eq (List a) where
  type xs == ys = Equals xs ys
  (%==) = sEquals

  eqRefl SNil         = Refl
  eqRefl (SCons x xs) = case (eqRefl x, eqRefl xs) of
    (Refl, Refl) -> Refl

  ...
```

Type-level-программирование и синглтоны

- Работает

Type-level-программирование и синглтоны

- Работает
- Но не стоит использовать (по крайней мере, в нынешнем виде)

Type-level-программирование и синглтоны

- Работает
- Но не стоит использовать (по крайней мере, в нынешнем виде)
  - Хотя, Data.Singletons.Prelude.* могут быть перспективными

Type-level-программирование и синглтоны

- Работает
- Но не стоит использовать (по крайней мере, в нынешнем виде)
  - Хотя, Data.Singletons.Prelude.* могут быть перспективными
- Приколы с ленивостью и отрицаниями

Type-level-программирование и синглтоны

- Работает
- Но не стоит использовать (по крайней мере, в нынешнем виде)
  - Хотя, Data.Singletons.Prelude.* могут быть перспективными
- Приколы с ленивостью и отрицаниями
- Вообще, изначально первоапрельская шутка[3]

[3] https://blog.jle.im/entry/verified-instances-in-haskell.html

Type-level-программирование и синглтоны

- Работает
- Но не стоит использовать (по крайней мере, в нынешнем виде)
  - Хотя, Data.Singletons.Prelude.* могут быть перспективными
- Приколы с ленивостью и отрицаниями
- Вообще, изначально первоапрельская шутка[3]
- Первое приближение в зависимым типам

---
[3] https://blog.jle.im/entry/verified-instances-in-haskell.html

Раздел 4

Интуиционистская логика и соответствие Карри-Ховарда

Классическая и интуиционистская логика

альтерн.	аристотелевская	конструктивная

Классическая и интуиционистская логика

альтерн.	аристотелевская	конструктивная
$\vdash p$	истинность	построение (конструктивное доказательство)

Классическая и интуиционистская логика

альтерн.	аристотелевская	конструктивная
$\vdash p$	истинность	построение (конструктивное доказательство)
$p \implies q$	истинность q если p истинно	построение q если дан p

Классическая и интуиционистская логика

альтерн.	аристотелевская	конструктивная
$\vdash p$	истинность	построение (конструктивное доказательство)
$p \implies q$	истинность q если p истинно	построение q если дан p
$\exists x \cdot P(x)$	доказательство невозможности предиката P быть ложным для всех x	построение объекта x, удовлетворяющего предикату P

Классическая и интуиционистская логика

альтерн.	аристотелевская	конструктивная
$\vdash p$	истинность	построение (конструктивное доказательство)
$p \implies q$	истинность q если p истинно	построение q если дан p
$\exists x \cdot P(x)$	доказательство невозможности предиката P быть ложным для всех x	построение объекта x, удовлетворяющего предикату P
$\neg p$	p ложно	p не может быть построен

Классическая и интуиционистская логика

альтерн.	аристотелевская	конструктивная
$\vdash p$	истинность	построение (конструктивное доказательство)
$p \implies q$	истинность q если p истинно	построение q если дан p
$\exists x \cdot P(x)$	доказательство невозможности предиката P быть ложным для всех x	построение объекта x, удовлетворяющего предикату P
$\neg p$	p ложно	p не может быть построен (имея объект p мы можем разрушить мир)

Классическая и интуиционистская логика

альтерн.	аристотелевская	конструктивная
$\vdash p$	истинность	построение (конструктивное доказательство)
$p \implies q$	истинность q если p истинно	построение q если дан p
$\exists x \cdot P(x)$	доказательство невозможности предиката P быть ложным для всех x	построение объекта x, удовлетворяющего предикату P
$\neg p$	p ложно	p не может быть построен (имея объект p мы можем разрушить мир)
$\neg\neg p$	p истинно	p не может быть не построен

- **с 1904** Лёйтзен Брауэр потвергает сомнению доказательства с $p \vee \neg p$
- **1925:** Аренд Гейтинг (ученик Брауэра) и Андрей Николаевич Колмогоров открывают способ формального построения сложных конструктивных доказательств из простых
- **1932-33,** Алонзо Чёрч: нетипизированное λ-исчисление
- **1934:** Хаскелл Карри (ученик Гильберта): соответствие утверждений импликативной логики с “типами” функций
- **1940,** Алонзо Чёрч: типизированное λ-исчисление (парадокс Клини-Россера)
- **1945:** Стивен Клини (ученик Чёрча): теория реализуемости, формализация сути конструктивных доказательств
- **1969:** Уильям Ховард (ученик МакЛейна): типы, соответствующие по Карри утверждениям первого порядка
- **к 1970** Николас Де Брёйн (независимо): proof assistant Automath
- **в 1971-79** Пер Мартин-Лёф (ученик Колмогорова): система типов, полиморфное λ-исчисление + зависимые типы

- **1988:** Thierry Coquand: CoC, лёгший основу Coq
- **ок. 2010:** Владимир Воеводский инициирует создание HoTT
- **2015:** Валерий Исаев: Arend, theorem prover на HoTT

Интуиционистская логика

Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`

Интуиционистская логика

Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`
$a \vee b$	`Either a b`

Интуиционистская логика

Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`
$a \vee b$	`Either a b`
$\top$	`Unit`

Интуиционистская логика
Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`
$a \vee b$	`Either a b`
$\top$	`Unit`
$\bot$	`Void`

Интуиционистская логика
Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`
$a \vee b$	`Either a b`
$\top$	`Unit`
$\bot$	`Void`
$a \Rightarrow b$	`a → b`

Интуиционистская логика

Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`
$a \vee b$	`Either a b`
$\top$	`Unit`
$\bot$	`Void`
$a \Rightarrow b$	`a → b`
$\neg a$	`a → Void`

Интуиционистская логика первого порядка

Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`
$a \vee b$	`Either a b`
$\top$	`Unit`
$\bot$	`Void`
$a \Rightarrow b$	`a → b`
$\neg a$	`a → Void`
$\forall x \colon a \cdot f(x)$	

Интуиционистская логика первого порядка

Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`
$a \vee b$	`Either a b`
$\top$	`Unit`
$\bot$	`Void`
$a \Rightarrow b$	`a → b`
$\neg a$	`a → Void`
$\forall x\colon a \cdot f(x)$	`(x : a) → f x`

Интуиционистская логика первого порядка

Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`
$a \vee b$	`Either a b`
$\top$	`Unit`
$\bot$	`Void`
$a \Rightarrow b$	`a → b`
$\neg a$	`a → Void`
$\forall x \colon a \cdot f(x)$	`(x : a) → f x`
$\exists x \colon a \cdot f(x)$	

Интуиционистская логика первого порядка

Соответствие Карри-Ховарда

$a \wedge b$	`(a, b)`
$a \vee b$	`Either a b`
$\top$	`Unit`
$\bot$	`Void`
$a \Rightarrow b$	`a → b`
$\neg a$	`a → Void`
$\forall x\colon a \cdot f(x)$	`(x : a) → f x`
$\exists x\colon a \cdot f(x)$	`(x : a ** f x)`

Интуиционистская логика первого порядка

Не неожиданные свойства

$a \wedge \bot \rightleftharpoons \bot$	`(a, Void)` $\rightleftharpoons$ `Void`

Интуиционистская логика первого порядка

Не неожиданные свойства

$a \wedge \bot \rightleftharpoons \bot$	`(a, Void)` $\rightleftharpoons$ `Void`
$a \wedge \top \rightleftharpoons a$	`(a, Unit)` $\rightleftharpoons$ `a`

Интуиционистская логика первого порядка

Не неожиданные свойства

$a \wedge \bot \rightleftharpoons \bot$	`(a, Void)` $\rightleftharpoons$ `Void`
$a \wedge \top \rightleftharpoons a$	`(a, Unit)` $\rightleftharpoons$ `a`
$a \vee \bot \rightleftharpoons a$	`Either a Void` $\rightleftharpoons$ `a`

Интуиционистская логика первого порядка

Не неожиданные свойства

$a \wedge \perp \rightleftharpoons \perp$	`(a, Void)` $\rightleftharpoons$ `Void`
$a \wedge \top \rightleftharpoons a$	`(a, Unit)` $\rightleftharpoons$ `a`
$a \vee \perp \rightleftharpoons a$	`Either a Void` $\rightleftharpoons$ `a`
$a \vee \top \rightleftharpoons \top$	`Either a Unit` $\rightleftharpoons$ `Unit`

Интуиционистская логика первого порядка

Не неожиданные свойства

$a \wedge \bot \rightleftharpoons \bot$	`(a, Void)` $\rightleftharpoons$ `Void`
$a \wedge \top \rightleftharpoons a$	`(a, Unit)` $\rightleftharpoons$ `a`
$a \vee \bot \rightleftharpoons a$	`Either a Void` $\rightleftharpoons$ `a`
$a \vee \top \rightleftharpoons \top$	`Either a Unit` $\rightleftharpoons$ `Unit`
$(a \wedge b \Rightarrow c) \rightleftharpoons (a \Rightarrow b \Rightarrow c)$	`(a, b) → c` $\rightleftharpoons$ `a → b → c`

Интуиционистская логика первого порядка

Не неожиданные свойства

$a \wedge \bot \rightleftharpoons \bot$	`(a, Void) ⇌ Void`
$a \wedge \top \rightleftharpoons a$	`(a, Unit) ⇌ a`
$a \vee \bot \rightleftharpoons a$	`Either a Void ⇌ a`
$a \vee \top \rightleftharpoons \top$	`Either a Unit ⇌ Unit`
$(a \wedge b \Rightarrow c) \rightleftharpoons (a \Rightarrow b \Rightarrow c)$	`(a, b) → c ⇌ a → b → c`
$((\exists x \colon a \cdot P(x)) \Rightarrow c) \rightleftharpoons (\forall x \colon a \cdot P(x) \Rightarrow c)$	`(x:a ** p x) → c ⇌ (x:a) → p x → c`

Интуиционистская логика первого порядка

Не неожиданные свойства

$a \wedge \bot \rightleftharpoons \bot$	`(a, Void)` $\rightleftharpoons$ `Void`
$a \wedge \top \rightleftharpoons a$	`(a, Unit)` $\rightleftharpoons$ `a`
$a \vee \bot \rightleftharpoons a$	`Either a Void` $\rightleftharpoons$ `a`
$a \vee \top \rightleftharpoons \top$	`Either a Unit` $\rightleftharpoons$ `Unit`
$(a \wedge b \Rightarrow c) \rightleftharpoons (a \Rightarrow b \Rightarrow c)$	`(a, b) → c` $\rightleftharpoons$ `a → b → c`
$((\exists x\colon a \cdot P(x)) \Rightarrow c)$ $\rightleftharpoons (\forall x\colon a \cdot P(x) \Rightarrow c)$	`(x:a ** p x) → c` $\rightleftharpoons$ `(x:a) → p x → c`
$\neg\neg\neg a \rightleftharpoons \neg a$	`((a → Void) → Void) → Void` $\rightleftharpoons$ `a → Void`

Интуиционистская логика первого порядка

Непривычно односторонние следствия

$\neg a \vee \neg b \Rightarrow \neg(a \wedge b)$

```
Either (a → Void) (b → Void) →
  (a, b) → Void
```

Интуиционистская логика первого порядка

Непривычно односторонние следствия

$\neg a \vee \neg b \Rightarrow \neg(a \wedge b)$

```
Either (a → Void) (b → Void) →
  (a, b) → Void
```

$a \wedge b \Rightarrow \neg(\neg a \vee \neg b)$

```
(a, b) →
  Either (a→Void)(b→Void)→Void
```

Интуиционистская логика первого порядка

Непривычно односторонние следствия

$\neg a \vee \neg b \Rightarrow \neg(a \wedge b)$

```
Either (a → Void) (b → Void) →
  (a, b) → Void
```

$a \wedge b \Rightarrow \neg(\neg a \vee \neg b)$

```
(a, b) →
  Either (a→Void)(b→Void)→Void
```

$\neg(\exists x \colon a \cdot P(x)) \Rightarrow \forall x \colon a \cdot \neg P(x)$

```
((x:a ** p x) → Void) →
  (y:a) → p y → Void
```

Интуиционистская логика первого порядка

Непривычно односторонние следствия

$\neg a \vee \neg b \Rightarrow \neg(a \wedge b)$	`Either (a → Void) (b → Void) →` `  (a, b) → Void`
$a \wedge b \Rightarrow \neg(\neg a \vee \neg b)$	`(a, b) →` `  Either (a→Void)(b→Void)→Void`
$\neg(\exists x : a \cdot P(x)) \Rightarrow \forall x : a \cdot \neg P(x)$	`((x:a ** p x) → Void) →` `  (y:a) → p y → Void`
$p \Rightarrow \neg\neg p$	`p →` `  (p → Void) → Void`

Вы уже пользуетесь зависимыми типами!

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
```

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
```

```
prepend :: forall a. a → [a] → [a]
```

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
prepend :: forall a. a → [a] → [a]
prepend :: forall (a :: *). a → [a] → [a]
```

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
prepend :: forall a. a → [a] → [a]
prepend :: forall (a :: *). a → [a] → [a]
prepend :: forall (a :: *). forall (x :: a). [a] → [a]
```

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
prepend :: forall a. a → [a] → [a]
prepend :: forall (a :: *). a → [a] → [a]
prepend :: forall (a :: *). forall (x :: a). [a] → [a]
prepend :: forall (a :: Type). a → [a] → [a]
```

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
prepend :: forall a. a → [a] → [a]
prepend :: forall (a :: *). a → [a] → [a]
prepend :: forall (a :: *). forall (x :: a). [a] → [a]
prepend :: forall (a :: Type). a → [a] → [a]
prepend :: (a :: Type) → a → [a] → [a]
```

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
prepend :: forall a. a → [a] → [a]
prepend :: forall (a :: *). a → [a] → [a]
prepend :: forall (a :: *). forall (x :: a). [a] → [a]
prepend :: forall (a :: Type). a → [a] → [a]
prepend :: (a :: Type) → a → [a] → [a]
prepend :: (a :: Type) → a → List a → List a
```

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
prepend :: forall a. a → [a] → [a]
prepend :: forall (a :: *). a → [a] → [a]
prepend :: forall (a :: *). forall (x :: a). [a] → [a]
prepend :: forall (a :: Type). a → [a] → [a]
prepend :: (a :: Type) → a → [a] → [a]
prepend :: (a :: Type) → a → List a → List a
prepend : (a : Type) → a → List a → List a
```

Вы уже пользуетесь зависимыми типами…

… с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
prepend :: forall a. a → [a] → [a]
prepend :: forall (a :: *). a → [a] → [a]
prepend :: forall (a :: *). forall (x :: a). [a] → [a]
prepend :: forall (a :: Type). a → [a] → [a]
prepend :: (a :: Type) → a → [a] → [a]
prepend :: (a :: Type) → a → List a → List a
prepend : (a : Type) → a → List a → List a
prepend : {a : Type} → a → List a → List a
```

Вы уже пользуетесь зависимыми типами...

... с тех пор, как узнали про параметрический полиморфизм

```
prepend :: a → [a] → [a]
prepend :: forall a. a → [a] → [a]
prepend :: forall (a :: *). a → [a] → [a]
prepend :: forall (a :: *). forall (x :: a). [a] → [a]
prepend :: forall (a :: Type). a → [a] → [a]
prepend :: (a :: Type) → a → [a] → [a]
prepend :: (a :: Type) → a → List a → List a
prepend : (a : Type) → a → List a → List a
prepend : {a : Type} → a → List a → List a
prepend : a → List a → List a
```

Раздел 5

Зависимые типы в действии

Тайпклассы “в законе”: dependent type edition

```
%default total
%access public export
```

Тайпклассы “в законе”: dependent type edition

```
%default total
%access public export

infix 6 ==, /=, <, <=, >, >=
```

Тайпклассы “в законе”: dependent type edition

```
%default total
%access public export

infix 6 ==, /=, <, <=, >, >=

interface Eq a where
  (==) : a -> a -> Bool

  eqRefl  : (x : a) -> So (x == x)
  eqSymm  : (x, y : a) -> x == y = y == x
  eqTrans : (x, y, z : a) -> So (x==y) -> So (y==z) -> So (x==z)

  (/=) : a -> a -> Bool
  x /= y = not $ x == y

  neqIsNotEq : (x, y : a) -> x == y = not (x /= y)
```

```
interface Eq a ⇒ Ord a where
  (<) : a → a → Bool

  ltArefl : (x : a) → So (not $ x < x)
  ltAsymm : (x, y : a) → So (x < y) → So (not $ y < x)
  ltTrans : (x, y, z : a) → So (x < y) → So (y < z) → So (x < z)

  (≤) : a → a → Bool
  x ≤ y = x < y || x == y
  lteIsLtOrE : (x, y : a) → x ≤ y = (x < y || x == y)

  (>) : a → a → Bool
  x > y = y < x
  gtInverseOfLt : (x, y : a) → x < y = y > x

  (≥) : a → a → Bool
  x ≥ y = y ≤ x
  gteIsGtOrE : (x, y : a) → x ≥ y = (x > y || x == y)
```

Реализации тайпклассов

```
Eq Nat where
```

Реализации тайпклассов

```
Eq Nat where
    Z     == Z     = True
    (S n) == (S m) = n == m
    _     == _     = False
```

Реализации тайпклассов с доказательствами законов

```
Eq Nat where
    Z     == Z     = True
    (S n) == (S m) = n == m
    _     == _     = False

    eqRefl Z     = Oh
    eqRefl (S n) = eqRefl n

    eqSymm Z     Z     = Refl
    eqSymm Z     (S _) = Refl
    eqSymm (S _) Z     = Refl
    eqSymm (S n) (S m) = eqSymm n m

    eqTrans Z     Z     Z     Oh Oh = Oh
    eqTrans (S i) (S j) (S k) p  q  = eqTrans i j k p q

    neqIsNotEq n m with (n == m)
      | True  = Refl
      | False = Refl
```

```
Ord Nat where
    Z     < (S _) = True
    (S n) < (S m) = n < m
    Z     < Z     = False
    (S _) < Z     = False
```

```
Ord Nat where
    Z     < (S _) = True
    (S n) < (S m) = n < m
    Z     < Z     = False
    (S _) < Z     = False

    ltArefl Z     = Oh
    ltArefl (S k) = ltArefl k

    ltAsymm Z     Z     _ = Oh
    ltAsymm Z     (S _) _ = Oh
    ltAsymm (S k) (S j) p = ltAsymm k j p

    ltTrans Z     (S _) (S _) Oh _ = Oh
    ltTrans (S i) (S j) (S k) p  q = ltTrans i j k p q

    lteIsLtOrE _ _ = Refl
    gtInverseOfLt _ _ = Refl

    gteIsGtOrE n m = rewrite eqSymm n m in Refl
```

Законные списки

```
(&&) : So p → So q → So (p && q)
Oh && Oh = Oh

soSplit : So (p && q) → (So p, So q)
soSplit {p=True} Oh = (Oh, Oh)

Eq a ⇒ Eq (List a) where
  []      == []      = True
  (x::xs) == (y::ys) = x == y && xs == ys
  _       == _       = False

  eqRefl []      = Oh
  eqRefl (x::xs) = eqRefl x && eqRefl xs

  ...
```

```
Eq a ⇒ Eq (List a) where
  ...

  eqSymm []        []        = Refl
  eqSymm []        (_ :: _)  = Refl
  eqSymm (_ :: _)  []        = Refl
  eqSymm (x :: xs) (y :: ys) = rewrite eqSymm x y in
                               rewrite eqSymm xs ys in
                               Refl

  eqTrans []        []        []        _ _ = Oh
  eqTrans (x :: xs) (y :: ys) (z :: zs) p q with (soSplit p, soSplit q)
    | ((xy, xys), (yz, yzs)) =
      eqTrans x y z xy yz && eqTrans xs ys zs xys yzs

  neqIsNotEq xs ys with (xs == ys)
    | True  = Refl
    | False = Refl
```

```
Ord a ⇒ Ord (List a) where
  []       < (_ :: _) = True
  _        < []       = False
  (x :: _) < (y :: _) = x < y

  ltArefl []       = Oh
  ltArefl (x :: _) = ltArefl x

  ltAsymm []       []       _ = Oh
  ltAsymm []       (_ :: _) _ = Oh
  ltAsymm (x :: _) (y :: _) p = ltAsymm x y p

  ltTrans []       (_ :: _) (_ :: _) _ _ = Oh
  ltTrans (x :: _) (y :: _) (z :: _) p q = ltTrans x y z p q

  lteIsLtOrE _ _ = Refl
  gtInverseOfLt _ _ = Refl

  gteIsGtOrE n m = rewrite eqSymm n m in Refl
```

Высшие порядки

```
interface Functor (f : Type → Type) where
  map : (a → b) → f a → f b

  mapPreservesId : (m : f a) → map (\x ⇒ x) m = m
  mapComposes : (m : f a) → (ab : a → b) → (bc : b → c)
                → map (bc . ab) m = map bc (map ab m)
```

Высшие порядки

```
interface Functor (f : Type → Type) where
  map : (a → b) → f a → f b

  mapPreservesId : (m : f a) → map (\x ⇒ x) m = m
  mapComposes : (m : f a) → (ab : a → b) → (bc : b → c)
              → map (bc . ab) m = map bc (map ab m)


Functor List where
  map f []        = []
  map f (x::xs) = f x :: map f xs

  mapPreservesId []        = Refl
  mapPreservesId (x::xs) = rewrite mapPreservesId xs in Refl

  mapComposes []        f g = Refl
  mapComposes (x::xs) f g = rewrite mapComposes xs f g in Refl
```

Раздел 6

Типизированные дырки

Разработка с постоянной компилируемостью

```
interface Eq a where
  ...
  eqSymm : (x, y : a) → x == y = y == x
  ...
```

Разработка с постоянной компилируемостью

```
    interface Eq a where
      ...
      eqSymm : (x, y : a) → x == y = y == x
      ...

Eq a ⇒ Eq (List a) where
  ...
  eqSymm xs ys = ?eqSymm_rhs
  ...
```

Диалог с компилятором

```
interface Eq a where
  ...
  eqSymm : (x, y : a) → x == y = y == x
  ...
```

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm xs ys = ?eqSymm_rhs
  ...
```

```
a : Type
xs : List a
ys : List a
constraint : Eq a
--------------------------------------
eqSymm_rhs : xs == ys = ys == xs
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []        []        = ?eqSymm_rhs_1
  eqSymm []        (y :: ys) = ?eqSymm_rhs_2
  eqSymm (x :: xs) []        = ?eqSymm_rhs_3
  eqSymm (x :: xs) (y :: ys) = ?eqSymm_rhs_4
  ...
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []        []        = ?eqSymm_rhs_1
  eqSymm []        (y :: ys) = ?eqSymm_rhs_2
  eqSymm (x :: xs) []        = ?eqSymm_rhs_3
  eqSymm (x :: xs) (y :: ys) = ?eqSymm_rhs_4
  ...
```

```
a : Type
constraint : Eq a
--------------------------------------
eqSymm_rhs_1 : True = True
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []        []        = Refl
  eqSymm []        (y :: ys) = ?eqSymm_rhs_2
  eqSymm (x :: xs) []        = ?eqSymm_rhs_3
  eqSymm (x :: xs) (y :: ys) = ?eqSymm_rhs_4
  ...
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
   ...
   eqSymm []        []        = Refl
   eqSymm []        (y :: ys) = ?eqSymm_rhs_2
   eqSymm (x :: xs) []        = ?eqSymm_rhs_3
   eqSymm (x :: xs) (y :: ys) = ?eqSymm_rhs_4
   ...
```

```
a : Type
y : a
ys : List a
constraint : Eq a
--------------------------------------
eqSymm_rhs_2 : False = False
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []        []        = Refl
  eqSymm []        (_ :: _)  = Refl
  eqSymm (x :: xs) []        = ?eqSymm_rhs_3
  eqSymm (x :: xs) (y :: ys) = ?eqSymm_rhs_4
  ...
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
   ...
  eqSymm []        []        = Refl
  eqSymm []        (_ :: _)  = Refl
  eqSymm (x :: xs) []        = ?eqSymm_rhs_3
  eqSymm (x :: xs) (y :: ys) = ?eqSymm_rhs_4
   ...
```

```
a : Type
x : a
xs : List a
constraint : Eq a
--------------------------------------
eqSymm_rhs_3 : False = False
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []        []        = Refl
  eqSymm []        (_ :: _)  = Refl
  eqSymm (_ :: _)  []        = Refl
  eqSymm (x :: xs) (y :: ys) = ?eqSymm_rhs_4
  ...
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []       []       = Refl
  eqSymm []       (_ :: _) = Refl
  eqSymm (_ :: _) []       = Refl
  eqSymm (x::xs) (y::ys) = ?eqSymm_rhs_4
  ...
```

```
a : Type
x : a
xs : List a
y : a
ys : List a
constraint : Eq a
--------------------------------------
eqSymm_rhs_4 : x == y && Delay (xs == ys) =
               y == x && Delay (ys == xs)
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  …
  eqSymm []        []        = Refl
  eqSymm []        (_ :: _)  = Refl
  eqSymm (_ :: _)  []        = Refl
  eqSymm (x :: xs) (y :: ys) = rewrite eqSymm x y in
                               ?eqSymm_rhs_4
  …
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []       []       = Refl
  eqSymm []       (_ :: _) = Refl
  eqSymm (_ :: _) []       = Refl
  eqSymm (x::xs) (y::ys) = rewrite eqSymm x y in
                            ?eqSymm_rhs_4
  ...
```

```
a : Type
x : a
xs : List a
y : a
ys : List a
constraint : Eq a
_rewrite_rule : x == y = y == x
--------------------------------------
eqSymm_rhs_4 : y == x && Delay (xs == ys) =
               y == x && Delay (ys == xs)
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []       (_ :: _)  = Refl
  eqSymm (_ :: _)  []       = Refl
  eqSymm (x :: xs) (y :: ys) = rewrite eqSymm x y in
                               rewrite eqSymm xs ys in
                               ?eqSymm_rhs_4
  ...
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []       (_ :: _)  = Refl
  eqSymm (_ :: _) []        = Refl
  eqSymm (x::xs) (y::ys) = rewrite eqSymm x y in
                           rewrite eqSymm xs ys in
                           ?eqSymm_rhs_4
  ...
```

```
a : Type
x : a
xs : List a
y : a
ys : List a
constraint : Eq a
_rewrite_rule : x == y = y == x
_rewrite_rule1 : xs == ys = ys == xs
--------------------------------------
eqSymm_rhs_4 : y == x && Delay (ys == xs) =
               y == x && Delay (ys == xs)
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []        []        = Refl
  eqSymm []        (_ :: _)  = Refl
  eqSymm (_ :: _)  []        = Refl
  eqSymm (x :: xs) (y :: ys) = rewrite eqSymm x y in
                               rewrite eqSymm xs ys in
                               Refl
  ...
```

Диалог с компилятором

```
Eq a ⇒ Eq (List a) where
  ...
  eqSymm []        []        = Refl
  eqSymm []        (_ :: _)  = Refl
  eqSymm (_ :: _)  []        = Refl
  eqSymm (x :: xs) (y :: ys) = rewrite eqSymm x y in
                               rewrite eqSymm xs ys in
                               Refl
  ...
```

Typechecks!

Раздел 7

Структурные ограничения

Тотальные нетотальные функции

```
%default total

at : (xs : List a) → (n : Nat) → a                               -- ???
```

Тотальные нетотальные функции

```
%default total

at : (xs : List a) → (n : Nat) → a                                -- ???


at : (xs : List a) → (n : Nat) → Maybe a
at []        _     = Nothing
at (x :: _)  Z     = Just x
at (_ :: xs) (S n) = xs `at` n
```

Аргументы-ограничения

```
%default total

at : (xs : List a) → (n : Nat) → a                              -- ???


at : (xs : List a) → (n : Nat) → Maybe a
at []         _     = Nothing
at (x :: _)   Z     = Just x
at (_ :: xs) (S n) = xs `at` n


at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a
```

Аргументы-ограничения

```
%default total

at : (xs : List a) → (n : Nat) → a                           -- ???


at : (xs : List a) → (n : Nat) → Maybe a
at []        _     = Nothing
at (x :: _)  Z     = Just x
at (_ :: xs) (S n) = xs `at` n


at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a

at (x :: _)  Z          = x
at (_ :: xs) (S k) {ok} = xs `at` k
```

Аргументы-ограничения

```
%default total

at : (xs : List a) → (n : Nat) → a                          -- ???


at : (xs : List a) → (n : Nat) → Maybe a
at []       _     = Nothing
at (x :: _) Z     = Just x
at (_ :: xs) (S n) = xs `at` n


at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a

at (x :: _) Z          = x
at (_ :: xs) (S k) {ok} = xs `at` k

at []       Z     impossible
at []       (S k) impossible
```

Аргументы-ограничения

```
at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a
```

Аргументы-ограничения

```
at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2
```

Аргументы-ограничения

```
at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2

x3 : Char
x3 = ['1', '2', '3'] `at` 3
```

Аргументы-ограничения

```
at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2

x3 : Char
x3 = ['1', '2', '3'] `at` 3
```

```
30 |   x3 = ['1', '2', '3'] `at` 3
   |                        ~~~~
When checking right hand side of x3 with expected type
        Char

When checking argument ok to function at:
        Can't find a value of type
                So False
```

Ограничения

```
at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a
at : (xs:List a) → (n:Nat) → {auto ok : So (n < length xs)} → a
```

Ограничения — специальные предикаты

```
at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a
at : (xs:List a) → (n:Nat) → {auto ok : So (n < length xs)} → a
```

```
data LTE : (n, m : Nat) → Type where
  LTEZero :                LTE Z     r
  LTESucc : LTE l r → LTE (S l) (S r)

total LT : Nat → Nat → Type
LT l r = LTE (S l) r
```

Ограничения — специальные предикаты

```
at : (xs:List a) → (n:Nat) → {auto ok:So (n `lt` length xs)} → a
at : (xs:List a) → (n:Nat) → {auto ok : So (n < length xs)} → a

   data LTE : (n, m : Nat) → Type where
     LTEZero :               LTE Z     r
     LTESucc : LTE l r → LTE (S l) (S r)

   total LT : Nat → Nat → Type
   LT l r = LTE (S l) r


at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a
at (x :: _)  Z                            = x
at (_ :: xs) (S n) {ok = (LTESucc _)} = xs `at` n
```

Ограничения — специальные предикаты

```
at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a
```

Ограничения — специальные предикаты

```
at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2

x3 : Char
x3 = ['1', '2', '3'] `at` 3
```

Ограничения — специальные предикаты

```
at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2

x3 : Char
x3 = ['1', '2', '3'] `at` 3
```

```
68 |   x3 = ['1', '2', '3'] `at` 3
   |                        ~~~~
When checking right hand side of x3 with expected type
        Char

When checking argument ok to function at:
        Can't find a value of type
                LTE 4 3
```

Ограничения — специальные предикаты

```
data LTE : (n, m : Nat) → Type where
  LTEZero :              LTE Z     r
  LTESucc : LTE l r → LTE (S l) (S r)
```

```
at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a
```

Ограничения — ещё более специальные предикаты

```
data LTE : (n, m : Nat) → Type where
  LTEZero :                 LTE Z     r
  LTESucc : LTE l r → LTE (S l) (S r)

at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a

data InBounds : (k : Nat) → (xs : List a) → Type where
    InFirst :                      InBounds Z     (x :: xs)
    InLater : InBounds k xs → InBounds (S k) (x :: xs)
```

Ограничения — ещё более специальные предикаты

```
    data LTE : (n, m : Nat) → Type where
      LTEZero :               LTE Z     r
      LTESucc : LTE l r → LTE (S l) (S r)

at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a


    data InBounds : (k : Nat) → (xs : List a) → Type where
        InFirst :                   InBounds Z     (x :: xs)
        InLater : InBounds k xs → InBounds (S k) (x :: xs)

at : (xs : List a) → (n : Nat) → {auto ok : InBounds n xs} → a
at (x :: _)  Z                         = x
at (_ :: xs) (S k) {ok = InLater _} = xs `at` k
```

Ограничения — ещё более специальные предикаты

```
at : (xs : List a) → (n : Nat) → {auto ok : InBounds n xs} → a
```

Ограничения — ещё более специальные предикаты

```
at : (xs : List a) → (n : Nat) → {auto ok : InBounds n xs} → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2

x3 : Char
x3 = ['1', '2', '3'] `at` 3
```

Ограничения — ещё более специальные предикаты

```
at : (xs : List a) → (n : Nat) → {auto ok : InBounds n xs} → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2

x3 : Char
x3 = ['1', '2', '3'] `at` 3
```

```
68 |   x3 = ['1', '2', '3'] `at` 3
   |                        ~~~~
When checking right hand side of x3 with expected type
        Char

When checking argument ok to function at:
        Can't find a value of type
                InBounds 3 ['1', '2', '3']
```

Ограничения в самом типе аргумента?

```
at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a
```

Ограничения в самом типе аргумента

```
at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a
```

```
data BoundedNat : Nat → Type where
  MkBNat : (n : Nat) → {auto ok : LT n b} → BoundedNat b
```

Ограничения в самом типе аргумента

```
at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a
```

```
data BoundedNat : Nat → Type where
  MkBNat : (n : Nat) → {auto ok : LT n b} → BoundedNat b
```

```
at : (xs : List a) → BoundedNat (length xs) → a
```

Ограничения в самом типе аргумента

```
at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a
```

```
data BoundedNat : Nat → Type where
  MkBNat : (n : Nat) → {auto ok : LT n b} → BoundedNat b
```

```
at : (xs : List a) → BoundedNat (length xs) → a

at (x :: _)  (MkBNat Z)                       = x
at (_ :: xs) (MkBNat (S n) {ok = (LTESucc _)}) = xs `at` MkBNat n
at []        (MkBNat n) impossible
```

Ограничения в самом типе аргумента

```
at : (xs : List a) → BoundedNat (length xs) → a
```

Ограничения в самом типе аргумента

```
at : (xs : List a) → BoundedNat (length xs) → a

x0 : Char
x0 = ['1', '2', '3'] `at` MkBNat 0

x2 : Char
x2 = ['1', '2', '3'] `at` MkBNat 2
```

Ограничения в самом типе аргумента

```
at : (xs : List a) → BoundedNat (length xs) → a

x0 : Char
x0 = ['1', '2', '3'] `at` MkBNat 0

x2 : Char
x2 = ['1', '2', '3'] `at` MkBNat 2

x3 : Char
x3 = ['1', '2', '3'] `at` MkBNat 3
```

Ограничения в самом типе аргумента

```
at : (xs : List a) → BoundedNat (length xs) → a

x0 : Char
x0 = ['1', '2', '3'] `at` MkBNat 0

x2 : Char
x2 = ['1', '2', '3'] `at` MkBNat 2

x3 : Char
x3 = ['1', '2', '3'] `at` MkBNat 3
```

```
87 |   x3 = ['1', '2', '3'] `at` MkBNat 3
   |                             ~~~~~~~~
When checking right hand side of x3 with expected type
        Char

When checking argument ok to constructor MkBNat:
        Can't find a value of type
                LTE 4 3
```

Неструктурные ограничения

```
data BoundedNat : Nat → Type where
  MkBNat : (n : Nat) → {auto ok : LT n b} → BoundedNat b
```

Структурные ограничения

```
data BoundedNat : Nat → Type where
  MkBNat : (n : Nat) → {auto ok : LT n b} → BoundedNat b
```

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

Структурные ограничения

```
data BoundedNat : Nat → Type where
  MkBNat : (n : Nat) → {auto ok : LT n b} → BoundedNat b
```

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

```
at : (xs : List a) → Fin (length xs) → a
at (x :: _)  FZ     = x
at (_ :: xs) (FS n) = xs `at` n
```

Структурные ограничения

```
at : (xs : List a) → Fin (length xs) → a
```

Структурные ограничения

```
at : (xs : List a) → Fin (length xs) → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2
```

Структурные ограничения

```
at : (xs : List a) → Fin (length xs) → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2

x3 : Char
x3 = ['1', '2', '3'] `at` 3
```

Структурные ограничения

```
at : (xs : List a) → Fin (length xs) → a

x0 : Char
x0 = ['1', '2', '3'] `at` 0

x2 : Char
x2 = ['1', '2', '3'] `at` 2

x3 : Char
x3 = ['1', '2', '3'] `at` 3
```

```
102 |   x3 = ['1', '2', '3'] `at` 3
    |                             ^
When checking right hand side of x3 with expected type
        Char

When checking argument prf to function Data.Fin.fromInteger:
        When using 3 as a literal for a Fin 3
                3 is not strictly less than 3
```

Зоопарк возможностей

Общие ограничения

```
at : (xs : List a) → (n : Nat) → {auto ok:So(n`lt`length xs)} → a
at : (xs : List a) → (n : Nat) → {auto ok:So (n < length xs)} → a
```

Ограничения со структурным предикатом

```
at : (xs : List a) → (n : Nat) → {auto ok : LT n (length xs)} → a
at : (xs : List a) → BoundedNat (length xs) → a
```

Структурные ограничения

```
at : (xs : List a) → (n : Nat) → {auto ok : InBounds n xs} → a
at : (xs : List a) → Fin (length xs) → a
```

Раздел 8

На что ещё способны зависимые типы

Типобезопасность и контекст

Пусть хотим[4] функцию sprintf

- на входе: строка с %-параметрами
- %d для целых, %f для плавающей запятой, %c для char'ов
- %% для самого символа "%"
- произвольное количество %-параметров
- типобезопасность (type-safety)

[4] https://www.codewars.com/kata/5c7fe5c859036f142eccaabb

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                              -- даёт "foobar"
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                                    -- даёт "foobar"
sprintf "foo%d" 5                                   -- даёт "foo5"
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                     -- даёт "foobar"
sprintf "foo%d" 5                    -- даёт "foo5"
sprintf "%coo%d%d" 'z' 6 5           -- даёт "zoo65"
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                     -- даёт "foobar"
sprintf "foo%d" 5                    -- даёт "foo5"
sprintf "%coo%d%d" 'z' 6 5           -- даёт "zoo65"
sprintf "%f%%%d" 1 0                 -- даёт "1.0%0"
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                    -- даёт "foobar"
sprintf "foo%d" 5                   -- даёт "foo5"
sprintf "%coo%d%d" 'z' 6 5          -- даёт "zoo65"
sprintf "%f%%%d" 1 0                -- даёт "1.0%0"
sprintf "%f%%%%d" 1                 -- даёт "1.0%%"
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                      -- даёт "foobar"
sprintf "foo%d" 5                     -- даёт "foo5"
sprintf "%coo%d%d" 'z' 6 5            -- даёт "zoo65"
sprintf "%f%%%d" 1 0                  -- даёт "1.0%0"
sprintf "%f%%%%d" 1                   -- даёт "1.0%%"
```

Не компилируются

```
sprintf "foo" 5                       -- лишний аргумент
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                      -- даёт "foobar"
sprintf "foo%d" 5                     -- даёт "foo5"
sprintf "%coo%d%d" 'z' 6 5            -- даёт "zoo65"
sprintf "%f%%%d" 1 0                  -- даёт "1.0%0"
sprintf "%f%%%%d" 1                   -- даёт "1.0%%"
```

Не компилируются

```
sprintf "foo" 5                       -- лишний аргумент
sprintf "foo%d" 'd'                   -- char, ожидался int
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                    -- даёт "foobar"
sprintf "foo%d" 5                   -- даёт "foo5"
sprintf "%coo%d%d" 'z' 6 5          -- даёт "zoo65"
sprintf "%f%%%d" 1 0                -- даёт "1.0%0"
sprintf "%f%%%%d" 1                 -- даёт "1.0%%"
```

Не компилируются

```
sprintf "foo" 5                     -- лишний аргумент
sprintf "foo%d" 'd'                 -- char, ожидался int
sprintf "foo%c" "d"                 -- string, ожидался char
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                    -- даёт "foobar"
sprintf "foo%d" 5                   -- даёт "foo5"
sprintf "%coo%d%d" 'z' 6 5          -- даёт "zoo65"
sprintf "%f%%%d" 1 0                -- даёт "1.0%0"
sprintf "%f%%%%d" 1                 -- даёт "1.0%%"
```

Не компилируются

```
sprintf "foo" 5                     -- лишний аргумент
sprintf "foo%d" 'd'                 -- char, ожидался int
sprintf "foo%c" "d"                 -- string, ожидался char
sprintf "foo%d" 4.0                 -- float, ожидался int
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                    -- даёт "foobar"
sprintf "foo%d" 5                   -- даёт "foo5"
sprintf "%coo%d%d" 'z' 6 5          -- даёт "zoo65"
sprintf "%f%%%d" 1 0                -- даёт "1.0%0"
sprintf "%f%%%%d" 1                 -- даёт "1.0%%"
```

Не компилируются

```
sprintf "foo" 5                     -- лишний аргумент
sprintf "foo%d" 'd'                 -- char, ожидался int
sprintf "foo%c" "d"                 -- string, ожидался char
sprintf "foo%d" 4.0                 -- float, ожидался int

putStrLn $ sprintf "fo%xo" x
putStrLn $ sprintf "fo%xo"
```

Типобезопасность и контекст

Компилируются

```
sprintf "foobar"                  -- даёт "foobar"
sprintf "foo%d" 5                 -- даёт "foo5"
sprintf "%coo%d%d" 'z' 6 5        -- даёт "zoo65"
sprintf "%f%%%d" 1 0              -- даёт "1.0%0"
sprintf "%f%%%%d" 1               -- даёт "1.0%%"
```

Не компилируются

```
sprintf "foo" 5                   -- лишний аргумент
sprintf "foo%d" 'd'               -- char, ожидался int
sprintf "foo%c" "d"               -- string, ожидался char
sprintf "foo%d" 4.0               -- float, ожидался int
```

```
putStrLn $ sprintf "fo%xo" x
putStrLn $ sprintf "fo%xo"
putStrLn $ sprintf "foo%" x
putStrLn $ sprintf "foo%"
```

Типобезопасность и контекст

```
export total
sprintf : (str : String) → SprintfType (unpack str)
```

Типобезопасность и контекст

```
    export total
    sprintf : (str : String) → SprintfType (unpack str)



public export total
SprintfType : List Char → Type
```

Типобезопасность и контекст

```
    export total
    sprintf : (str : String) → SprintfType (unpack str)



public export total
SprintfType : List Char → Type
SprintfType []                     =           String
```

Типобезопасность и контекст

```
export total
sprintf : (str : String) → SprintfType (unpack str)
```

```
public export total
SprintfType : List Char → Type
SprintfType []                  =            String
SprintfType ('%' :: '%' :: rest) =           SprintfType rest
```

Типобезопасность и контекст

```
export total
sprintf : (str : String) → SprintfType (unpack str)
```

```
public export total
SprintfType : List Char → Type
SprintfType []                  =            String
SprintfType ('%' :: '%' :: rest) =            SprintfType rest
SprintfType ('%' :: 'd' :: rest) = ?sprintfType_rhs_d
```

Типобезопасность и контекст

```
    export total
    sprintf : (str : String) → SprintfType (unpack str)
```

```
public export total
SprintfType : List Char → Type
SprintfType []                  =                      String
SprintfType ('%' :: '%' :: rest) =                     SprintfType rest
SprintfType ('%' :: 'd' :: rest) = Integer → SprintfType rest
```

Типобезопасность и контекст

```
export total
sprintf : (str : String) → SprintfType (unpack str)
```

```
public export total
SprintfType : List Char → Type
SprintfType []                  =                String
SprintfType ('%' :: '%' :: rest) =               SprintfType rest
SprintfType ('%' :: 'd' :: rest) = Integer →     SprintfType rest
SprintfType ('%' :: 'f' :: rest) = Double  →     SprintfType rest
```

Типобезопасность и контекст

```
export total
sprintf : (str : String) → SprintfType (unpack str)
```

```
public export total
SprintfType : List Char → Type
SprintfType []                   =            String
SprintfType ('%' :: '%' :: rest) =            SprintfType rest
SprintfType ('%' :: 'd' :: rest) = Integer → SprintfType rest
SprintfType ('%' :: 'f' :: rest) = Double  → SprintfType rest
SprintfType ('%' :: 'c' :: rest) = Char    → SprintfType rest
```

Типобезопасность и контекст

```
export total
sprintf : (str : String) → SprintfType (unpack str)
```

```
public export total
SprintfType : List Char → Type
SprintfType []                  =              String
SprintfType ('%' :: '%' :: rest) =              SprintfType rest
SprintfType ('%' :: 'd' :: rest) = Integer →   SprintfType rest
SprintfType ('%' :: 'f' :: rest) = Double  →   SprintfType rest
SprintfType ('%' :: 'c' :: rest) = Char    →   SprintfType rest
SprintfType ('%' :: rest)        = ?sprintfType_rhs_bad
```

Типобезопасность и контекст

```
export total
sprintf : (str : String) → SprintfType (unpack str)
```

```
public export total
SprintfType : List Char → Type
SprintfType []                 =            String
SprintfType ('%' :: '%' :: rest) =            SprintfType rest
SprintfType ('%' :: 'd' :: rest) = Integer → SprintfType rest
SprintfType ('%' :: 'f' :: rest) = Double  → SprintfType rest
SprintfType ('%' :: 'c' :: rest) = Char    → SprintfType rest
SprintfType ('%' :: rest)        = Void    → SprintfType rest
```

Типобезопасность и контекст

```
export total
sprintf : (str : String) → SprintfType (unpack str)
```

```
public export total
SprintfType : List Char → Type
SprintfType []                 =            String
SprintfType ('%' :: '%' :: rest) =            SprintfType rest
SprintfType ('%' :: 'd' :: rest) = Integer → SprintfType rest
SprintfType ('%' :: 'f' :: rest) = Double  → SprintfType rest
SprintfType ('%' :: 'c' :: rest) = Char    → SprintfType rest
SprintfType ('%' :: rest)        = Void    → SprintfType rest
SprintfType ( _ :: rest)         =            SprintfType rest
```

Типобезопасность и контекст

```
public export total
st : (curr : String) → (str : List Char) → (t : Type ** t)
```

Типобезопасность и контекст

```
public export total
st : (curr : String) → (str : List Char) → (t : Type ** t)
st c []                 = (_ ** c)
```

Типобезопасность и контекст

```
public export total
st : (curr : String) → (str : List Char) → (t : Type ** t)
st c []                 = (_ ** c)
st c ('%'::'%'::ks) = st (c ++ "%") ks
```

Типобезопасность и контекст

```
public export total
st : (curr : String) → (str : List Char) → (t : Type ** t)
st c []                 = (_ ** c)
st c ('%'::'%'::ks) = st (c ++ "%") ks
st c ('%'::'d'::ks) = (_ ** \n:Integer⇒snd $ st (c++show n) ks)
```

Типобезопасность и контекст

```
public export total
st : (curr : String) → (str : List Char) → (t : Type ** t)
st c []                = (_ ** c)
st c ('%'::'%'::ks) = st (c ++ "%") ks
st c ('%'::'d'::ks) = (_ ** \n:Integer⇒snd $ st (c++show n) ks)
st c ('%'::'f'::ks) = (_ ** \x:Double ⇒snd $ st (c++show x) ks)
```

Типобезопасность и контекст

```
public export total
st : (curr : String) → (str : List Char) → (t : Type ** t)
st c []                  = (_ ** c)
st c ('%'::'%'::ks) = st (c ++ "%") ks
st c ('%'::'d'::ks) = (_ ** \n:Integer⇒snd $ st (c++show n) ks)
st c ('%'::'f'::ks) = (_ ** \x:Double ⇒snd $ st (c++show x) ks)
st c ('%'::'c'::ks) = (_ ** \k:Char   ⇒snd$ st (c++singleton k) ks)
```

Типобезопасность и контекст

```
public export total
st : (curr : String) → (str : List Char) → (t : Type ** t)
st c []                    = (_ ** c)
st c ('%'::'%'::ks) = st (c ++ "%") ks
st c ('%'::'d'::ks) = (_ ** \n:Integer⇒snd $ st (c++show n) ks)
st c ('%'::'f'::ks) = (_ ** \x:Double ⇒snd $ st (c++show x) ks)
st c ('%'::'c'::ks) = (_ ** \k:Char   ⇒snd$ st (c++singleton k) ks)
st c ('%':: ks)       = (_ ** \v:Void   ⇒snd $ st c ks)
```

Типобезопасность и контекст

```
public export total
st : (curr : String) → (str : List Char) → (t : Type ** t)
st c []                 = (_ ** c)
st c ('%'::'%'::ks) = st (c ++ "%") ks
st c ('%'::'d'::ks) = (_ ** \n:Integer⇒snd $ st (c++show n) ks)
st c ('%'::'f'::ks) = (_ ** \x:Double ⇒snd $ st (c++show x) ks)
st c ('%'::'c'::ks) = (_ ** \k:Char   ⇒snd$ st (c++singleton k) ks)
st c ('%':: ks)       = (_ ** \v:Void   ⇒snd $ st c ks)
st c ( k :: ks)       = st (c ++ singleton k) ks
```

Типобезопасность и контекст

```
public export total
st : (curr : String) → (str : List Char) → (t : Type ** t)
st c []                 = (_ ** c)
st c ('%'::'%'::ks) = st (c ++ "%") ks
st c ('%'::'d'::ks) = (_ ** \n:Integer⇒snd $ st (c++show n) ks)
st c ('%'::'f'::ks) = (_ ** \x:Double ⇒snd $ st (c++show x) ks)
st c ('%'::'c'::ks) = (_ ** \k:Char   ⇒snd$ st (c++singleton k) ks)
st c ('%':: ks)     = (_ ** \v:Void   ⇒snd $ st c ks)
st c ( k :: ks)     = st (c ++ singleton k) ks


export total
sprintf : (str : String) → fst $ st "" $ unpack str
sprintf str = snd $ st "" $ unpack str
```

Состояние в типе

```
data Access = LoggedOut | LoggedIn
data LoginResult = OK | BadPassword

interface DataStore (m : Type → Type) where
  Store : Access → Type

  connect : ST m Var [add (Store LoggedOut)]
  disconnect : (store : Var) → ST m () [remove store (Store LoggedOut)]

  login : (store : Var) →
          ST m LoginResult [store ::: Store LoggedOut :→
                               (\res ⇒ Store (case res of
                                                    OK ⇒ LoggedIn
                                                    BadPassword ⇒ LoggedOut))]
  logout : (store : Var) → ST m () [store ::: Store LoggedIn :→ Store LoggedOut]

  readSecret : (store : Var) → ST m String [store ::: Store LoggedIn]
```

[4] http://docs.idris-lang.org/en/latest/st/machines.html

Раздел 9

Заключение

Итог

Итог

Дороги трудны, но хуже без дорог
Ю. Визбор

Итог

Дороги трудны, но хуже без дорог
Ю. Визбор

Зависимые типы

- столь же фундаментальны, как полиморфизм и type constructor

Итог

Дороги трудны, но хуже без дорог
Ю. Визбор

Зависимые типы

- столь же фундаментальны, как полиморфизм и type constructor
- открывают неимоверные возможности

Итог

Дороги трудны, но хуже без дорог
Ю. Визбор

Зависимые типы

- столь же фундаментальны, как полиморфизм и type constructor
- открывают неимоверные возможности
- требуют аккуратности в обращении

Итог

Дороги трудны, но хуже без дорог
Ю. Визбор

Зависимые типы

- столь же фундаментальны, как полиморфизм и type constructor
- открывают неимоверные возможности
- требуют аккуратности в обращении
- прекрасно сочетаются с другими системами

Итог?

Дороги трудны, но хуже без дорог
Ю. Визбор

Зависимые типы

- столь же фундаментальны, как полиморфизм и type constructor
- открывают неимоверные возможности
- требуют аккуратности в обращении
- прекрасно сочетаются с другими системами
- мы стоим в середине пути

Итог?

Дороги трудны, но хуже без дорог
Ю. Визбор

Зависимые типы

- столь же фундаментальны, как полиморфизм и type constructor
- открывают неимоверные возможности
- требуют аккуратности в обращении
- прекрасно сочетаются с другими системами
- мы стоим в середине пути

В субботу, 25-го, Виталий Брагилевский с докладом
Dependent types: is this the future we want for out programming languages?

Вдохновение

- Philip Wadler, Propositions as Types (2015)
  - канонический доклад (лямбда-супергерой)
  - статья
  - ещё доклад и ещё
- Edwin Brady, Type-driven Development of Communicating Systems in Idris (2016)
- Edwin Brady, Idris 2 - Type-driven Development of Idris (2018)
- Edwin Brady, Type driven Development in Action (2019)
- Николай Николаевич Непейвода, курс про логику (2008), см. книгу Прикладная логика

элементы списка нажимабельны

Спасибо

Вопросы?