

Циферки в типах
или
Quantitative type theory

Чудесное настоящее и прекрасное будущее

Денис Буздалов

03.02.2021

Глава 1

Теория типов?

Дисклеймеры

Дисклеймеры

- По содержанию
 - ▶ Область, о которой речь, развивается с большой скоростью

Дисклеймеры

- По содержанию
 - ▶ Область, о которой речь, развивается с большой скоростью
 - ▶ Целью доклада не ставится глубинное и исчерпывающее изложение изложение теории

Дисклеймеры

- По содержанию
 - ▶ Область, о которой речь, развивается с большой скоростью
 - ▶ Целью доклада не ставится глубинное и исчерпывающее изложение изложение теории
 - ▶ Цель доклада показать как вообще бывает, если вы с таким не сталкивались показать как *ещё* бывает, если вы с чем-то таким сталкивались

Дисклеймеры

- По содержанию
 - ▶ Область, о которой речь, развивается с большой скоростью
 - ▶ Целью доклада не ставится глубинное и исчерпывающее изложение изложение теории
 - ▶ Цель доклада показать как вообще бывает, если вы с таким не сталкивались показать как *ещё* бывает, если вы с чем-то таким сталкивались
- По оформлению
 - ▶ Будет очень много кода

Дисклеймеры

- По содержанию
 - ▶ Область, о которой речь, развивается с большой скоростью
 - ▶ Целью доклада не ставится глубинное и исчерпывающее изложение изложение теории
 - ▶ Цель доклада показать как вообще бывает, если вы с таким не сталкивались показать как *ещё* бывает, если вы с чем-то таким сталкивались
- По оформлению
 - ▶ Будет очень много кода
 - ▶ Плавно появляющийся материал и оверлеи

- Типизированные языки программирования

- Типизированные языки программирования
 - ▶ Типы как выразительное средство
 - ▶ Типы как помощь для *правильной* реализации

- Типизированные языки программирования
 - ▶ Типы как выразительное средство
 - ▶ Типы как помощь для *правильной* реализации
- QTT — именно теория, не относится к конкретному языку

- Типизированные языки программирования
 - ▶ Типы как выразительное средство
 - ▶ Типы как помощь для *правильной* реализации
- QTT — именно теория, не относится к конкретному языку
 - ▶ На ней и её вариациях уже построено несколько языков
Idris, Granule, Juvix, Gerty

- Типизированные языки программирования
 - ▶ Типы как выразительное средство
 - ▶ Типы как помощь для *правильной* реализации
- QTT — именно теория, не относится к конкретному языку
 - ▶ На ней и её вариациях уже построено несколько языков
Idris, Granule, Juvix, Gerty
 - ▶ Не обязательно с зависимыми типами

- Типизированные языки программирования
 - ▶ Типы как выразительное средство
 - ▶ Типы как помощь для *правильной* реализации
- QTT — именно теория, не относится к конкретному языку
 - ▶ На ней и её вариациях уже построено несколько языков
Idris, Granule, Juvix, Gerty
 - ▶ Не обязательно с зависимыми типами
 - ▶ Не обязательно функциональные

- Типизированные языки программирования
 - ▶ Типы как выразительное средство
 - ▶ Типы как помощь для *правильной* реализации
- QTT — именно теория, не относится к конкретному языку
 - ▶ На ней и её вариациях уже построено несколько языков
Idris, Granule, Juvix, Gerty
 - ▶ Не обязательно с зависимыми типами
 - ▶ Не обязательно функциональные
 - ▶ Частный случай: линейные типы
Rust, Linear Haskell, Cogent, ...

Стечения обстоятельств рассмотрения (волюнтаризмы)

- Язык для демонстраций
 - ▶ чисто функциональный

Стечения обстоятельств рассмотрения (волюнтаризмы)

- Язык для демонстраций
 - ▶ чисто функциональный
 - ▶ с зависимыми типами

Стечения обстоятельств рассмотрения (волюнтаризмы)

- Язык для демонстраций
 - ▶ чисто функциональный
 - ▶ с зависимыми типами
 - ▶ синтаксис Idris 2 с вольными расширениями

Стечения обстоятельств рассмотрения (волюнтаризмы)

- Язык для демонстраций
 - ▶ чисто функциональный
 - ▶ с зависимыми типами
 - ▶ синтаксис Idris 2 с вольными расширениями
- Параметризация QTТ
 - ▶ в самом Idris 2 частный случай

Стечения обстоятельств рассмотрения (волюнтаризмы)

- Язык для демонстраций
 - ▶ чисто функциональный
 - ▶ с зависимыми типами
 - ▶ синтаксис Idris 2 с вольными расширениями
- Параметризация QTТ
 - ▶ в самом Idris 2 частный случай
 - ▶ будут примеры более общей параметризации

Стечения обстоятельств рассмотрения (волютаризмы)

- Язык для демонстраций
 - ▶ чисто функциональный
 - ▶ с зависимыми типами
 - ▶ синтаксис Idris 2 с вольными расширениями
- Параметризация QTT
 - ▶ в самом Idris 2 частный случай
 - ▶ будут примеры более общей параметризации
 - ▶ эти общие примеры *могут* не компилироваться

Стечения обстоятельств рассмотрения (волютаризмы)

- Язык для демонстраций
 - ▶ чисто функциональный
 - ▶ с зависимыми типами
 - ▶ синтаксис Idris 2 с вольными расширениями
- Параметризация QTТ
 - ▶ в самом Idris 2 частный случай
 - ▶ будут примеры более общей параметризации
 - ▶ эти общие примеры *могут* не компилироваться
- Специфичные для QTТ аннотации в коде могут принадлежать
 - ▶ стрелке функций ($a \rightarrow b$ vs. $a -o b$ или $a \%1 \rightarrow b$)
 - ▶ аргументам функций и полям структур данных
 - ▶ типу

Стечения обстоятельств рассмотрения (волютаризмы)

- Язык для демонстраций
 - ▶ чисто функциональный
 - ▶ с зависимыми типами
 - ▶ синтаксис Idris 2 с вольными расширениями
- Параметризация QTT
 - ▶ в самом Idris 2 частный случай
 - ▶ будут примеры более общей параметризации
 - ▶ эти общие примеры *могут* не компилироваться
- Специфичные для QTT аннотации в коде могут принадлежать
 - ▶ стрелке функций ($a \rightarrow b$ vs. $a -o b$ или $a \%1 \rightarrow b$)
 - ▶ **аргументам функций и полям структур данных**
 - ▶ типу

Стечения обстоятельств рассмотрения (волютаризмы)

- Язык для демонстраций
 - ▶ чисто функциональный
 - ▶ с зависимыми типами
 - ▶ синтаксис Idris 2 с вольными расширениями
- Параметризация QTT
 - ▶ в самом Idris 2 частный случай
 - ▶ будут примеры более общей параметризации
 - ▶ эти общие примеры *могут* не компилироваться
- Специфичные для QTT аннотации в коде могут принадлежать
 - ▶ стрелке функций ($a \rightarrow b$ vs. $a -o b$ или $a \%1 \rightarrow b$)
 - ▶ **аргументам функций и полям структур данных**
 - ▶ **типу**

Центральная мысль

- Точные (в частности, зависимые) типы — о том, *что* можно

Центральная мысль

- Точные (в частности, зависимые) типы — о том, *что* можно
- `head : (l : List a) → NonEmpty l ⇒ a`

Центральная мысль

- Точные (в частности, зависимые) типы — о том, *что* можно

`head : (l : List a) → NonEmpty l ⇒ a`

`index : Fin len → Vect len a → a`

Центральная мысль

- Точные (в частности, зависимые) типы — о том, *что* можно

`head` : (l : List a) → NonEmpty l ⇒ a

`index` : Fin len → Vect len a → a

`(++)` : Vect n a → Vect m a → Vect (n + m) a

Центральная мысль

- Точные (в частности, зависимые) типы — о том, *что* можно

`head` : `(l : List a) → NonEmpty l ⇒ a`

`index` : `Fin len → Vect len a → a`

`(++)` : `Vect n a → Vect m a → Vect (n + m) a`

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)`

Центральная мысль

- Точные (в частности, зависимые) типы — о том, *что* можно

`head` : `(l : List a) → NonEmpty l ⇒ a`

`index` : `Fin len → Vect len a → a`

`(++)` : `Vect n a → Vect m a → Vect (n + m) a`

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)`

- QTT — о том, *когда* можно

Глава 2

«Ноль»

Ограничения на параметры

- `head : (l : List a) → NonEmpty l ⇒ a`

Ограничения на параметры

- `head` : (`l` : `List a`) \rightarrow `NonEmpty l` \Rightarrow `a`
`head (x :: _) = x`

Ограничения на параметры

%default total

- `head : (l : List a) → NonEmpty l ⇒ a`
`head (x :: _) = x`

Ограничения на параметры

`%default total`

- `head : (l : List a) → NonEmpty l ⇒ a`
`head (x :: _) = x`
- `head : (l : List a) → {auto _ : NonEmpty l} → a`
`head (x :: _) = x`

%default total

- `head : (l : List a) → NonEmpty l ⇒ a`
`head (x :: _) = x`
- `head : (l : List a) → {auto _ : NonEmpty l} → a`
`head (x :: _) = x`

%default total

- `head : (l : List a) → NonEmpty l ⇒ a`
`head (x :: _) = x`
- `head : (l : List a) → {auto _ : NonEmpty l} → a`
`head (x :: _) = x`
- `head : (l : List a) → {auto 0 _ : NonEmpty l} → a`
`head (x :: _) = x`

%default total

- `head : (l : List a) → NonEmpty l ⇒ a`
`head (x :: _) = x`
- `head : (l : List a) → {auto _ : NonEmpty l} → a`
`head (x :: _) = x`
- `head : (l : List a) → {auto 0 _ : NonEmpty l} → a`
`head (x :: _) = x`
- `head : (l : List a) → (0 _ : NonEmpty l) ⇒ a`
`head (x :: _) = x`

%default total

- `head : (l : List a) → NonEmpty l ⇒ a`
`head (x :: _) = x`
- `head : (l : List a) → {auto _ : NonEmpty l} → a`
`head (x :: _) = x`
- `head : (l : List a) → {auto 0 _ : NonEmpty l} → a`
`head (x :: _) = x`
- `head : (l : List a) → (0 _ : NonEmpty l) ⇒ a`
`head (x :: _) = x`
- 0-параметры
 - ▶ можем рассуждать о них при компиляции
 - ▶ не можем использовать при выполнении

Явные “мета”-параметры

```
interface Monad m => MonadState stateType m | m where  
  get : m stateType  
  put : stateType -> m Unit
```


Явные “мета”-параметры

```
interface Monad m => MonadState stateType m | m where
  get : m stateType
  put : stateType -> m Unit
```

```
f : (MonadState Int m, MonadState Nat m, Monad m) => m Integer
f = put {- Int -} 0 *> map natToInteger get
```

Явные “мета”-параметры

```
interface Monad m => MonadState stateType m | m where
  get  : m stateType
  put  : stateType -> m Unit
```

```
f : (MonadState Int m, MonadState Nat m, Monad m) => m Integer
f = put {- Int -} 0 *> map natToInteger get
```

```
the : (a : Type) -> (x : a) -> a
the _ x = x
```

Явные “мета”-параметры

```
interface Monad m => MonadState stateType m | m where
  get  : m stateType
  put  : stateType -> m Unit
```

```
f : (MonadState Int m, MonadState Nat m, Monad m) => m Integer
f = put (the Int 0) *> map natToInteger get
```

```
the : (a : Type) -> (x : a) -> a
the _ x = x
```

```
interface Monad m => MonadState stateType m | m where
  get  : m stateType
  put  : stateType -> m Unit
```

```
f : (MonadState Int m, MonadState Nat m, Monad m) => m Integer
f = put (the Int 0) *> map natToInteger get
```

```
the : (a : Type) -> (x : a) -> a
the _ x = x
```

```
interface Monad m => MonadState stateType m | m where
  get  : m stateType
  put  : stateType -> m Unit
```

```
f : (MonadState Int m, MonadState Nat m, Monad m) => m Integer
f = put (the Int 0) *> map natToInteger get
```

```
the : (0 a : Type) -> (x : a) -> a
the _ x = x
```

Proof-carrying code

```
filt : (p : a → Bool) → List a → List (x : a ** p x = True)
```

Proof-carrying code

```
filt : (p : a → Bool) → List a → List (x : a ** p x = True)
```

```
record DPair a p where  
  constructor MkDPair  
  fst : a  
  snd : p fst
```

```
filt : (p : a → Bool) → List a → List (x : a ** p x = True)
```

```
record DPair a p where  
  constructor MkDPair  
  fst : a  
  snd : p fst
```



```
filt : (p : a → Bool) → List a → List (x : a ** p x = True)
```

```
record Subset a p where  
  constructor Element  
  fst : a  
  0 snd : p fst
```

Proof-carrying code

хорошо, когда нет в runtime

```
filt : (p : a → Bool) → List a → List (x : a ** p x = True)
```

```
record Subset a p where
  constructor Element
  fst : a
  0 snd : p fst
```

```
filt : (p : a → Bool) → List a → List $ Subset a (\x⇒p x = True)
```

```
filt : (p : a → Bool) → List a → List (x : a ** p x = True)
```

```
record Subset a p where
  constructor Element
  fst : a
  0 snd : p fst
```

```
filt : (p : a → Bool) → List a → List $ Subset a (\x⇒p x = True)
filt p [] = []
filt p (x::xs) = case @@ p x of
  (True ** prf) ⇒ Element x prf :: filt p xs
  (False ** _) ⇒ filt p xs
```

Parametricity

- $\text{id} : a \rightarrow a$
 $\text{id } x = x$

Parametricity

- $\text{id} : a \rightarrow a$
 $\text{id } x = x$

- ▶ Гордость Haskell'я: подобная сигнатура допускает единственную реализацию

Parametricity

- $\text{id} : a \rightarrow a$
 $\text{id } x = x$

▶ Гордость Haskell'я: подобная сигнатура допускает единственную реализацию

- $\text{id} : \text{forall } a. a \rightarrow a$
 $\text{id } x = x$

Parametricity

- $\text{id} : a \rightarrow a$
 $\text{id } x = x$

▶ Гордость Haskell'я: подобная сигнатура допускает единственную реализацию

- $\text{id} : \text{forall } a. a \rightarrow a$
 $\text{id } x = x$

- $\text{id} : \{a : \text{Type}\} \rightarrow a \rightarrow a$
 $\text{id } x = x$

Parametricity

- `id : a → a`
`id x = x`

▶ Гордость Haskell'я: подобная сигнатура допускает единственную реализацию

- `id : forall a. a → a`
`id x = x`

- `id : {a : Type} → a → a`
`id {a=Int} x = x + 1`
`id {a=String} x = x ++ "a"`
`id x = x`

- `id : a → a`

`id x = x`

- ▶ Гордость Haskell'я: подобная сигнатура допускает единственную реализацию

- `id : forall a. a → a`

`id x = x`

- `id : {a : Type} → a → a`

`id {a=Int} x = x + 1`

`id {a=String} x = x ++ "a"`

`id x = x`

- `id : a → a`
`id x = x`
 - ▶ Гордость Haskell'я: подобная сигнатура допускает единственную реализацию
- `id : forall a. a → a`
`id x = x`
- `id : {a : Type} → a → a`
`id {a=Int} x = x + 1`
`id {a=String} x = x ++ "a"`
`id x = x`
- `id : {0 a : Type} → a → a`
`id x = x`

- $id : a \rightarrow a$

`id x = x`

- ▶ Гордость Haskell'я: подобная сигнатура допускает единственную реализацию

- $id : forall a. a \rightarrow a$

`id x = x`

- $id : \{a : Type\} \rightarrow a \rightarrow a$

`id {a=Int} x = x + 1`

`id {a=String} x = x ++ "a"`

`id x = x`

- $id : \{0 a : Type\} \rightarrow a \rightarrow a$

`id x = x`

- ▶ На самом деле `forall a` — сахар для `{0 a : _}`

Когда ещё важно

Когда ещё важно

```
data Nat = Z | S Nat
```

Когда ещё важно

```
data Nat = Z | S Nat
```

- Удобно рассуждать
- Неэкономно в рантайме *

Когда ещё важно

```
data Nat = Z | S Nat
```

- Удобно рассуждать
- Неэкономно в рантайме *

```
data BNat : Nat → Type where  
  BZ : BNat 0  
  B0 : BNat n → BNat (2*n)  
  B1 : BNat n → BNat (1 + 2*n)
```

Когда ещё важно

```
data Nat = Z | S Nat
```

- Удобно рассуждать
- Неэкономно в рантайме *

```
data BNat : Nat → Type where
```

```
  BZ : BNat 0
```

```
  B0 : BNat n → BNat (2*n)
```

```
  B1 : BNat n → BNat (1 + 2*n)
```

```
(+) : BNat n → BNat m → BNat (n + m)
```


Когда ещё важно

```
data Nat = Z | S Nat
```

- Удобно рассуждать
- Неэкономно в рантайме *

```
data BNat : Nat → Type where
```

```
  BZ : BNat 0
```

```
  B0 : BNat n → BNat (2*n)
```

```
  B1 : BNat n → BNat (1 + 2*n)
```

```
(+) : BNat n → BNat m → BNat (n + m)
```

```
toNat : BNat n → Subset Nat (\k ⇒ k = n)
```

Когда ещё важно

```
data Nat = Z | S Nat
```

- Удобно рассуждать
- Неэкономно в рантайме *

```
data BNat : Nat → Type where
```

```
  BZ : BNat 0
```

```
  B0 : BNat n → BNat (2*n)
```

```
  B1 : BNat n → BNat (1 + 2*n)
```

```
(+) : BNat n → BNat m → BNat (n + m)
```

```
toNat : BNat n → Subset Nat (\k ⇒ k = n)
```

```
toNat x = Element n Refl
```

Когда ещё важно

```
data Nat = Z | S Nat
```

- Удобно рассуждать
- Неэкономно в рантайме *

```
data BNat : Nat → Type where
```

```
  BZ : BNat 0
```

```
  B0 : BNat n → BNat (2*n)
```

```
  B1 : BNat n → BNat (1 + 2*n)
```

```
(+) : BNat n → BNat m → BNat (n + m)
```

```
toNat : {n : Nat} → BNat n → Subset Nat (\k ⇒ k = n)
```

```
toNat x = Element n Refl
```

Когда ещё важно

```
data Nat = Z | S Nat
```

- Удобно рассуждать
- Неэкономно в рантайме *

```
data BNat : Nat → Type where
```

```
  BZ : BNat 0
```

```
  B0 : BNat n → BNat (2*n)
```

```
  B1 : BNat n → BNat (1 + 2*n)
```

```
(+) : BNat n → BNat m → BNat (n + m)
```

```
toNat : BNat n → Subset Nat (\k ⇒ k = n)
```

```
toNat BZ      = Element 0 Refl
```

```
toNat (B0 x) = let Element s prf = toNat x in  
                Element (2*s) rewrite prf in Refl
```

```
toNat (B1 x) = let Element s prf = toNat x in  
                Element (1 + 2*s) rewrite prf in Refl
```

Нет в runtime — не значит нельзя матчить

Нет в runtime — не значит нельзя матчить

```
record Subset a p where
  constructor Element
  fst : a
  0 snd : p fst
```

-- напоминание --

Нет в runtime — не значит нельзя матчить

```
record Subset a p where -- напоминание --
  constructor Element
  fst : a
  () snd : p fst

data All : (() p : a → Type) → List a → Type where
  Nil : All p Nil
  (::) : {() xs : List a} → p x → All p xs → All p (x::xs)
```

Нет в runtime — не значит нельзя матчить

```
record Subset a p where                                     -- напоминание --
  constructor Element
  fst : a
  0 snd : p fst

data All : (0 p : a → Type) → List a → Type where
  Nil  : All p Nil
  (::)  : {0 xs : List a} → p x → All p xs → All p (x::xs)

pushIn : (xs : List a) → (0 _ : All p xs) → List $ Subset a p
```


Нет в runtime — не значит нельзя матчить

```
record Subset a p where -- напоминание --
  constructor Element
  fst : a
  () snd : p fst

data All : (() p : a → Type) → List a → Type where
  Nil : All p Nil
  (::) : {() xs : List a} → p x → All p xs → All p (x::xs)

pushIn : (xs : List a) → (() _ : All p xs) → List $ Subset a p
pushIn [] _ = []
```

Нет в runtime — не значит нельзя матчить

```
record Subset a p where                                     -- напоминание --
  constructor Element
  fst : a
  () snd : p fst

data All : (() p : a → Type) → List a → Type where
  Nil : All p Nil
  (::) : {() xs : List a} → p x → All p xs → All p (x::xs)

pushIn : (xs : List a) → (() _ : All p xs) → List $ Subset a p
pushIn [] _ = []
pushIn (x::xs) prf = ?pushIn_rhs
```

Нет в runtime — не значит нельзя матчить

```
record Subset a p where                                     -- напоминание --
  constructor Element
  fst : a
  () snd : p fst

data All : (() p : a → Type) → List a → Type where
  Nil : All p Nil
  (::) : {() xs : List a} → p x → All p xs → All p (x::xs)

pushIn : (xs : List a) → (() _ : All p xs) → List $ Subset a p
pushIn [] _ = []
pushIn (x::xs) prf = ?pushIn_rhs

-----
pushIn_rhs : List (Subset a p)
```

Нет в runtime — не значит нельзя матчить

```
record Subset a p where -- напоминание --
  constructor Element
  fst : a
  () snd : p fst

data All : (() p : a → Type) → List a → Type where
  Nil : All p Nil
  (::) : {() xs : List a} → p x → All p xs → All p (x::xs)

pushIn : (xs : List a) → (() _ : All p xs) → List $ Subset a p
pushIn [] _ = []
pushIn (x::xs) prf = Element x ?r :: pushIn xs ?rs
```

Нет в runtime — не значит нельзя матчить

```
record Subset a p where                                     -- напоминание --
  constructor Element
  fst : a
  () snd : p fst

data All : (() p : a → Type) → List a → Type where
  Nil : All p Nil
  (::) : {() xs : List a} → p x → All p xs → All p (x::xs)

pushIn : (xs : List a) → (() _ : All p xs) → List $ Subset a p
pushIn [] _ = []
pushIn (x::xs) prf = Element x ?r :: pushIn xs ?rs

-----
r : p x
rs : All p xs
```

Нет в runtime — не значит нельзя матчить

```
record Subset a p where -- напоминание --
  constructor Element
  fst : a
  () snd : p fst

data All : (() p : a → Type) → List a → Type where
  Nil : All p Nil
  (::) : {() xs : List a} → p x → All p xs → All p (x::xs)

pushIn : (xs : List a) → (() _ : All p xs) → List $ Subset a p
pushIn [] _ = []
pushIn (x::xs) (r::rs) = Element x r :: pushIn xs rs
```

Нет в runtime — не значит нельзя матчить

```
record Subset a p where                                     -- напоминание --
  constructor Element
  fst : a
  () snd : p fst

data All : (() p : a → Type) → List a → Type where
  Nil : All p Nil
  (::) : {() xs : List a} → p x → All p xs → All p (x::xs)

pushIn : (xs : List a) → (() _ : All p xs) → List $ Subset a p
pushIn [] [] = []
pushIn (x::xs) (r::rs) = Element x r :: pushIn xs rs
```

Когда?

Когда?

- 0
- ▶ никогда

Когда?

- 0
 - ▶ `никогда` в рантайме
 - ▶ неограниченно для рассуждений при компиляции

Глава 3

«Один»

Данные vs. ресурсы

- *Компиляторы* редко оперируют понятием *ресурса*

Данные vs. ресурсы

- *Компиляторы* редко оперируют понятием *ресурса*
- Данные обычно
 - ▶ неограниченно долго (много) используемые
 - ▶ неограниченно много копируемые
 - ▶ можно выбросить в любой момент

Данные vs. ресурсы

- *Компиляторы* редко оперируют понятием *ресурса*
- Данные обычно
 - ▶ неограниченно долго (много) используемые
 - ▶ неограниченно много копируемые
 - ▶ можно выбросить в любой момент
- Реальный мир так не устроен

Данные vs. ресурсы

- *Компиляторы* редко оперируют понятием *ресурса*
- Данные обычно
 - ▶ неограниченно долго (много) используемые
 - ▶ неограниченно много копируемые
 - ▶ можно выбросить в любой момент
- Реальный мир так не устроен
- Данными мы иногда моделируем вещи реального мира

Данные vs. ресурсы

- *Компиляторы* редко оперируют понятием *ресурса*
- Данные обычно
 - ▶ неограниченно долго (много) используемые
 - ▶ неограниченно много копируемые
 - ▶ можно выбросить в любой момент
- Реальный мир так не устроен
- Данными мы иногда моделируем вещи реального мира
- Это всё порой приводит к проблемам

Нельзя несколько

Нельзя несколько

- Вспомните *итераторы* из любимых вами языков

Нельзя несколько

- Помните *итераторы* из любимых вами языков
- Нельзя несколько раз удалить файл

Нельзя несколько

- Вспомните *итераторы* из любимых вами языков
- Нельзя несколько раз удалить файл
- Искусственный пример

```
data DisResult = CantDisconnect | Disconnected
data DisconnectGrant : Arm → Type where ...
```

Нельзя несколько

- Вспомните *итераторы* из любимых вами языков
- Нельзя несколько раз удалить файл
- Искусственный пример

```
data DisResult = CantDisconnect | Disconnected
data DisconnectGrant : Arm → Type where ...
```

```
wantDisconnect : (arm : Arm) → IO $ Maybe $ DisconnectGrant arm
disconnect : DisconnectGrant arm → IO Unit
```

Нельзя несколько

- Вспомните *итераторы* из любимых вами языков
- Нельзя несколько раз удалить файл
- Искусственный пример

```
data DisResult = CantDisconnect | Disconnected
data DisconnectGrant : Arm → Type where ...
```

```
wantDisconnect : (arm : Arm) → IO $ Maybe $ DisconnectGrant arm
disconnect : DisconnectGrant arm → IO Unit
```

```
whatever : IO DisResult
whatever = do
```

```
  Just rm ← wantDisconnect LeftTopArm
  | Nothing ⇒ pure CantDisconnect
  disconnect rm
  pure Disconnected
```

Нельзя выкидывать

- Пример: специальное обязательство для закрытия файла

```
data ClosingLia : File → Type where ...
```

Нельзя выкидывать

- Пример: специальное обязательство для закрытия файла

```
data ClosingLia : File → Type where ...
```

```
openFile  : (fl : File) → IO $ Maybe $ ClosingLia fl
```

```
readChar  : (fl : File) → (0 _ : ClosingLia fl) ⇒ IO Char
```

```
closeFile : ClosingLia h → IO Unit
```


Нельзя выкидывать

- Пример: специальное обязательство для закрытия файла

```
data ClosingLia : File → Type where ...
```

```
openFile : (fl : File) → IO $ Maybe $ ClosingLia fl
```

```
readChar : (fl : File) → (0 _ : ClosingLia fl) ⇒ IO Char
```

```
closeFile : ClosingLia h → IO Unit
```

```
whatever : File → IO $ Either String Char
```

```
whatever fl = do
```

```
  Just cl ← openFile fl
```

```
  | Nothing ⇒ pure $ Left "Can't open"
```

```
  c ← readChar fl
```

```
  closeFile cl
```

```
  pure $ Right c
```

Линейные типы

- Линейный параметр
 - ▶ используется в рантайме строго один раз

Линейные типы

- Линейный параметр
 - ▶ используется в рантайме строго один раз
нельзя копировать
нельзя **не** использовать

Линейные типы

- Линейный параметр

- ▶ используется в рантайме строго один раз...
нельзя копировать
нельзя **не** использовать
- ▶ ...если функция используется строго один раз

Линейные типы

- Линейный параметр
 - ▶ используется в рантайме строго один раз...
нельзя копировать
нельзя **не** использовать
 - ▶ ...если функция используется строго один раз
- “Использовать”
 - ▶ pattern match
 - ▶ вернуть
 - ▶ передать в функцию как параметр с ненулевым quantity

Возможности линейных типов

- Отражать в сигнатурах больше
- Отражать в сигнатурах новое
 - ▶ Ресурсы
 - ▶ Протоколы
- Оптимизация выполнения

Возможности линейных типов

- **Отражать в сигнатурах больше**
- Отражать в сигнатурах новое
 - ▶ Ресурсы
 - ▶ Протоколы
- Оптимизация выполнения

Более точные спецификации функций

```
(++) : List a → List a → List a  
[]      ++ ys = ?concat_rhs1  
(x :: xs) ++ ys = ?concat_rhs2
```


Более точные спецификации функций

```
(++) : (1 _ : List a) → (1 _ : List a) → List a  
[]      ++ ys = ?concat_rhs1  
(x :: xs) ++ ys = ?concat_rhs2
```

Более точные спецификации функций

```
data List : Type → Type where
```

```
  Nil  : List a
```

```
  (::)  : (1 _ : a) → (1 _ : List a) → List a
```

```
(++) : (1 _ : List a) → (1 _ : List a) → List a
```

```
[]      ++ ys = ?concat_rhs1
```

```
(x::xs) ++ ys = ?concat_rhs2
```

Более точные спецификации функций

```
data List : Type → Type where
```

```
  Nil  : List a
```

```
  (::)  : (1 _ : a) → (1 _ : List a) → List a
```

```
(++) : (1 _ : List a) → (1 _ : List a) → List a
```

```
[]      ++ ys = ?concat_rhs1
```

```
(x::xs) ++ ys = ?concat_rhs2
```

```
0 a : Type
```

```
1 ys : List a
```

```
concat_rhs1 : List a
```

Более точные спецификации функций

```
data List : Type → Type where
```

```
  Nil  : List a
```

```
  (::)  : (1 _ : a) → (1 _ : List a) → List a
```

```
(++) : (1 _ : List a) → (1 _ : List a) → List a
```

```
[] ++ ys = ys
```

```
(x::xs) ++ ys = ?concat_rhs2
```

Более точные спецификации функций

```
data List : Type → Type where
```

```
Nil      : List a
```

```
(::)     : (1 _ : a) → (1 _ : List a) → List a
```

```
(++)    : (1 _ : List a) → (1 _ : List a) → List a
```

```
[]      ++ ys = ys
```

```
(x::xs) ++ ys = ?concat_rhs2
```

```
0 a : Type
```

```
1 x : a
```

```
1 xs : List a
```

```
1 ys : List a
```

```
concat_rhs2 : List a
```

Более точные спецификации функций

```
data List : Type → Type where
```

```
  Nil  : List a
```

```
  (::)  : (1 _ : a) → (1 _ : List a) → List a
```

```
(++) : (1 _ : List a) → (1 _ : List a) → List a
```

```
[ ] ++ ys = ys
```

```
(x :: xs) ++ ys = x :: xs ++ ys
```

Не все данные одинаково полезны

```
data List : Type → Type where
  Nil    : List a
  (::)   : (1 _ : a) → (1 _ : List a) → List a
```

```
null : (1 _ : List a) → Bool
```

Не все данные одинаково полезны

```
data List : Type → Type where
  Nil  : List a
  (::)  : (1 _ : a) → (1 _ : List a) → List a
```

```
null : (1 _ : List a) → Bool
null []      = True
null (_::_) = False
```


Не все данные одинаково полезны

```
data List : Type → Type where
  Nil  : List a
  (::)  : (1 _ : a) → (1 _ : List a) → List a
```

```
null : List a → Bool
null []      = True
null (_ :: _) = False
```

Не все данные одинаково полезны

```
data List : Type → Type where
  Nil  : List a
  (::)  : a → List a → List a
```

```
null : (1 _ : List a) → Bool
null []      = True
null (_ :: _) = False
```

Не все данные одинаково полезны

```
data List : Type → Type where
  Nil  : List a
  (::)  : a → List a → List a
```

```
null : (1 _ : List a) → Bool
null []      = True
null (_ :: _) = False
```

- Фантазии:

Не все данные одинаково полезны

```
data List : Type → Type where
  Nil  : List a
  (::)  : (? _ : a) → (? _ : List a) → List a
```

```
null : (1 _ : List a) → Bool
null []      = True
null (_::_) = False
```

- Фантазии:
 - ▶ данные, полиморфные по quantity

Не все данные одинаково полезны

```
data List : Type → Type where
  Nil      : List a
  (::)     : (? _ : a) → (? _ : List a) → List a
```

```
null : (pm _ : List a) → Bool
null []      = True
null (_::_) = False
```

- Фантазии:

- ▶ данные, полиморфные по quantity
- ▶ промежуточная quantity, только на pattern matching: $0 < pm < 1$

Не все данные одинаково полезны

```
data List : Type → Type where
  Nil      : List a
  (::)     : (? _ : a) → (? _ : List a) → List a
```

```
null : (pm _ : List a) → Bool
null []      = True
null (_::_) = False
```

- Фантазии:
 - ▶ данные, полиморфные по quantity
 - ▶ промежуточная quantity, только на pattern matching: $0 < pm < 1$
- А что, например, с
 - ▶ `head : (l : List a) → (0 _ : NonEmpty l) ⇒ a`

Не все данные одинаково полезны

```
data List : Type → Type where
  Nil      : List a
  (::)     : (? _ : a) → (? _ : List a) → List a
```

```
null : (pm _ : List a) → Bool
null []           = True
null (_::_)      = False
```

- Фантазии:
 - ▶ данные, полиморфные по quantity
 - ▶ промежуточная quantity, только на pattern matching: $0 < pm < 1$
- А что, например, с
 - ▶ `head` : $(l : List a) \rightarrow (0 _ : NonEmpty l) \Rightarrow a$
 - ▶ `tail` : $(l : List a) \rightarrow (0 _ : NonEmpty l) \Rightarrow a$

Фантазируем дальше: очень точные спецификации

```
data List : Type → Type where
```

```
  Nil  : List a
```

```
  (::)  : (? x : a) → (? xs : List a) → List a
```

```
(++) : (l _ : List a) → (l _ : List a) → List a
```

```
null : (pm _ : List a) → Bool
```


Фантазируем дальше: очень точные спецификации

```
data List : Type → Type where
```

```
  Nil  : List a
```

```
  (::)  : (? x : a) → (? xs : List a) → List a
```

```
(++) : (1{x=1,xs=1} _ : List a) → (1{x=1,xs=1} _ : List a) → List a
```

```
null : (1{x=0,xs=0} _ : List a) → Bool
```

Фантазируем дальше: очень точные спецификации

```
data List : Type → Type where
```

```
  Nil  : List a
```

```
  (::)  : (? x : a) → (? xs : List a) → List a
```

```
(++) : (1{x=1,xs=1} _ : List a) → (1{x=1,xs=1} _ : List a) → List a
```

```
null : (1{x=0,xs=0} _ : List a) → Bool
```

```
head : (1{x=1,xs=0} l : List a) → (0 _ : NonEmpty l) ⇒ a
```

```
tail : (1{x=0,xs=1} l : List a) → (0 _ : NonEmpty l) ⇒ a
```

Фантазируем дальше: очень точные спецификации

```
data List : Type → Type where
```

```
  Nil  : List a
```

```
  (::) : (? x : a) → (? xs : List a) → List a
```

```
(++) : (1{_=1} _ : List a) → (1{_=1} _ : List a) → List a
```

```
null : (1{_=0} _ : List a) → Bool
```

```
head : (1{x=1,xs=0} l : List a) → (0 _ : NonEmpty l) ⇒ a
```

```
tail : (1{x=0,xs=1} l : List a) → (0 _ : NonEmpty l) ⇒ a
```

Фантазируем дальше: очень точные спецификации

```
data List : Type → Type where
```

```
  Nil : List a
```

```
  (::) : (? x : a) → (? xs : List a) → List a
```

```
(++) : (1 _ : List a) → (1 _ : List a) → List a
```

```
null : (1{_=0} _ : List a) → Bool
```

```
head : (1{x=1,xs=0} l : List a) → (0 _ : NonEmpty l) ⇒ a
```

```
tail : (1{x=0,xs=1} l : List a) → (0 _ : NonEmpty l) ⇒ a
```

Фантазируем дальше: очень точные спецификации

```
data List : Type → Type where
```

```
  Nil : List a
```

```
  (::) : (? x : a) → (? xs : List a) → List a
```

```
(++) : (1 _ : List a) → (1 _ : List a) → List a
```

```
null : (1{_=0} _ : List a) → Bool
```

```
head : (1{x=1,xs=0} l : List a) → (0 _ : NonEmpty l) ⇒ a
```

```
tail : (1{x=0,xs=1} l : List a) → (0 _ : NonEmpty l) ⇒ a
```

Философский вопрос: *нужно ли это?*

Псевдо-quantity-полиморфизм

```
data LinOrW = Linear | W
data List' : {l, r : LinOrW} → Type → Type where
  Nil      : List' a
  (::)     : a → List' a → List' {l=W,r=W} a
```

Псевдо-quantity-полиморфизм

```
data LinOrW = Linear | W
```

```
data List' : {l, r : LinOrW} → Type → Type where
```

```
Nil      : List' a
```

```
(::)     : a → List' a → List' {l=W,r=W} a
```

```
(.::)    : (1 _ : a) → List' a → List' {l=1,r=W} a
```

```
(::.)    : a → (1 _ : List' a) → List' {l=W,r=1} a
```

```
(.::.)   : (1 _ : a) → (1 _ : List' a) → List' {l=1,r=1} a
```

Псевдо-quantity-полиморфизм

```
data LinOrW = Linear | W
```

```
data List' : {l, r : LinOrW} → Type → Type where
```

```
  Nil      : List' a
```

```
  (::)     : a → List' a → List' {l=W,r=W} a
```

```
  (.:)    : (1 _ : a) → List' a → List' {l=1,r=W} a
```

```
  (::.)    : a → (1 _ : List' a) → List' {l=W,r=1} a
```

```
  (.:.)   : (1 _ : a) → (1 _ : List' a) → List' {l=1,r=1} a
```

```
relax : (1 _ : List' a) → List' {l=1,r=1} a
```


Псевдо-quantity-полиморфизм

```
data LinOrW = Linear | W
```

```
data List' : {l, r : LinOrW} → Type → Type where
```

```
  Nil      : List' a
```

```
  (::)     : a → List' a → List' {l=W,r=W} a
```

```
  (.:)    : (1 _ : a) → List' a → List' {l=1,r=W} a
```

```
  (::.)    : a → (1 _ : List' a) → List' {l=W,r=1} a
```

```
  (.:.)   : (1 _ : a) → (1 _ : List' a) → List' {l=1,r=1} a
```

```
relax : (1 _ : List' a) → List' {l=1,r=1} a
```

```
null : (1 _ : List' {l=W,r=W} a) → Bool
```

```
(++) : (1 _ : List' {l=1,r=1} a) → (1 _ : List' {l=1,r=1} a) →  
      List' {l=1,r=1} a
```

Псевдо-quantity-полиморфизм

```
data LinOrW = Linear | W
```

```
data List' : {l, r : LinOrW} → Type → Type where
```

```
  Nil      : List' a
```

```
  (::)     : a → List' a → List' {l=W,r=W} a
```

```
  (.:)    : (1 _ : a) → List' a → List' {l=1,r=W} a
```

```
  (::.)    : a → (1 _ : List' a) → List' {l=W,r=1} a
```

```
  (.:.)   : (1 _ : a) → (1 _ : List' a) → List' {l=1,r=1} a
```

```
relax : (1 _ : List' a) → List' {l=1,r=1} a
```

```
null : (1 _ : List' {l=W,r=W} a) → Bool
```

```
(++) : (1 _ : List' {l=1,r=1} a) → (1 _ : List' {l=1,r=1} a) →  
      List' {l=1,r=1} a
```

```
f : List' a → List' a → List' {l=1,r=1} a
```

```
f xs ys = relax xs ++ relax ys
```

Псевдо-quantity-полиморфизм

```
data LinOrW = Linear | W
```

```
data List' : {l, r : LinOrW} → Type → Type where
```

```
  Nil      : List' a
```

```
  (::)     : a → List' a → List' {l=W,r=W} a
```

```
  (.:)    : (1 _ : a) → List' a → List' {l=1,r=W} a
```

```
  (::.)    : a → (1 _ : List' a) → List' {l=W,r=1} a
```

```
  (.:.)   : (1 _ : a) → (1 _ : List' a) → List' {l=1,r=1} a
```

```
relax : (1 _ : List' a) → List' {l=1,r=1} a
```

```
null : (1 _ : List' {l=W,r=W} a) → Bool
```

```
(++) : (1 _ : List' {l=1,r=1} a) → (1 _ : List' {l=1,r=1} a) →  
      List' {l=1,r=1} a
```

```
f : List' a → List' a → List' {l=1,r=1} a
```

```
f xs ys = relax xs ++ relax ys
```

```
-- :-( --
```

Напоминание: возможности линейных типов

- Отражать в сигнатурах больше
- Отражать в сигнатурах новое
 - ▶ Ресурсы
 - ▶ Протоколы
- Оптимизация выполнения

Напоминание: возможности линейных типов

- Отражать в сигнатурах больше
- Отражать в сигнатурах новое
 - ▶ **Ресурсы**
 - ▶ Протоколы
- Оптимизация выполнения

Моделирование ресурсов, их жизненный цикл

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

1 create : Params → Resource

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

```
1 create : Params → Resource
```

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
```

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

```
1 create : Params → Resource
```

```
data Ur : Type → Type where
```

```
  MkUr : a → Ur a
```

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
```

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

```
1 create : Params → Resource
```

```
data Ur : Type → Type where
```

```
  MkUr : a → Ur a
```

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
```

- Потребление: нельзя использовать после

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

```
1 create : Params → Resource
```

```
data Ur : Type → Type where
```

```
  MkUr : a → Ur a
```

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
```

- Потребление: нельзя использовать после

```
destroy : (1 _ : Resource) → Ur Result -- или просто Unit
```

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

```
1 create : Params → Resource
```

```
data Ur : Type → Type where
```

```
  MkUr : a → Ur a
```

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
```

- Потребление: нельзя использовать после

```
destroy : (1 _ : Resource) → Ur Result -- или просто Unit
```

- Нерасходующие, но зависящие?

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

```
1 create : Params → Resource
```

```
data Ur : Type → Type where
```

```
  MkUr : a → Ur a
```

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
```

- Потребление: нельзя использовать после

```
destroy : (1 _ : Resource) → Ur Result -- или просто Unit
```

- Нерасходующие, но зависящие? Потребить+создать

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

```
1 create : Params → Resource
```

```
data Ur : Type → Type where
```

```
  MkUr : a → Ur a
```

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
```

- Потребление: нельзя использовать после

```
destroy : (1 _ : Resource) → Ur Result -- или просто Unit
```

- Нерасходуемые, но зависящие? Потребить+создать

```
depend : (1 _ : Resource) → LPair' Result Resource
```

Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

```
1 create : Params → Resource
```

```
data Ur : Type → Type where
```

```
  MkUr : a → Ur a
```

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
```

- Потребление: нельзя использовать после

```
destroy : (1 _ : Resource) → Ur Result -- или просто Unit
```

- Нерасходуемые, но зависящие? Потребить+создать

```
data LPair' : Type → Type → Type where
```

```
  (#) : a → (1 _ : b) → LPair' a b
```

```
depend : (1 _ : Resource) → LPair' Result Resource
```


Моделирование ресурсов, их жизненный цикл

- Создание: нельзя выкинуть (обязательно использовать)

```
1 create : Params → Resource
```

```
data Ur : Type → Type where
```

```
  MkUr : a → Ur a
```

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
```

- Потребление: нельзя использовать после

```
destroy : (1 _ : Resource) → Ur Result -- или просто Unit
```

- Нерасходуемые, но зависящие? Потребить+создать

```
data LPair' : Type → Type → Type where
```

```
  (#) : a → (1 _ : b) → LPair' a b
```

```
depend : (1 _ : Resource) → LPair' Result Resource
```

- Borrowing

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    MkUr ?foo
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    MkUr ?foo
```

1 res : Resource

foo : Result

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    MkUr ?foo
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    MkUr ?foo
```

```
0 res : Resource
  r : Result
1 res' : Resource
```

```
foo : Result
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let (s # res'') = depend res in
    MkUr ?foo
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let (s # res'') = depend res in
    MkUr ?foo
```

Error: While processing right hand side of f.
res is not accessible in this context.

... /Example/Qtt/One.idr:120:35--120:38

```
120 |           let (s # res'') = depend res in
      |                                     ^^^
```


Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let (s # res'') = depend res' in
    MkUr ?foo
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let (s # res'') = depend res' in
    MkUr ?foo
```

```
0 res : Resource
  r : Result
0 res' : Resource
  s : Result
1 res'' : Resource
```

```
foo : Result
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let _ = destroy res' in
    MkUr ?foo
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let _ = destroy res' in
    MkUr ?foo
```

```
∅ res : Resource
  r : Result
∅ res' : Resource
```

```
foo : Result
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let _ = destroy res' in
    MkUr r
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let _ = destroy res' in
    MkUr r
```

Error: While processing right hand side of f.
There are 0 uses of linear name res'.

... /Example/Qtt/One.idr:99:46--99:54

```
99 | depend : (1 _ : Resource) → LPair' Result Resource
    | ^^^^^^^^^
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let z = destroy res' in
    MkUr r
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let z = destroy res' in
    MkUr r
```

Error: While processing right hand side of f.
There are 0 uses of linear name z.

```
... /Example/QtT/One.idr:120:14--120:30
```

```
120 |           let z = destroy res' in
      |               ^^^^^^^^^^^^^^^^^
```


Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let () = destroy res' in
    MkUr r
```

Использование ресурса: вырожденный пример

```
runWithCreated : Params → (1 _ : (1 _ : Resource) → Ur a) → a
destroy : (1 _ : Resource) → Unit
depend : (1 _ : Resource) → LPair' Result Resource
```

```
f : Result
f = runWithCreated params \res ⇒
    let (r # res') = depend res in
    let () = destroy res' in
    MkUr r
```

Successfully reloaded .../Example/Qt/One.idr

Использование ресурса: пример с файлами и IO

```
data FileHandler : FileName → Type where ...
```

Использование ресурса: пример с файлами и IO

```
data FileHandler : FileName → Type where ...
```

```
withOpenFile : LinearIO io ⇒ (fn : FileName) →  
  (success : (1 _ : FileHandler fn) → L io a) →  
  (fail : L io a) → L io a
```

Использование ресурса: пример с файлами и IO

```
data FileHandler : FileName → Type where ...
```

```
withOpenFile : LinearIO io ⇒ (fn : FileName) →  
    (success : (1 _ : FileHandler fn) → L io a) →  
    (fail : L io a) → L io a
```

```
closeFile : LinearIO io ⇒ (1 _ : FileHandler fn) → L io Unit
```

Использование ресурса: пример с файлами и IO

```
data FileHandler : FileName → Type where ...
```

```
withOpenFile : LinearIO io ⇒ (fn : FileName) →  
    (success : (1 _ : FileHandler fn) → L io a) →  
    (fail : L io a) → L io a
```

```
closeFile : LinearIO io ⇒ (1 _ : FileHandler fn) → L io Unit
```

```
readLine : LinearIO io ⇒ (1 _ : FileHandler fn) →  
    L io {use=1} $ LPair' String $ FileHandler fn
```

Использование ресурса: пример с файлами и IO

```
data FileHandler : FileName → Type where ...
```

```
withOpenFile : LinearIO io ⇒ (fn : FileName) →  
    (success : (1 _ : FileHandler fn) → L io a) →  
    (fail : L io a) → L io a
```

```
closeFile : LinearIO io ⇒ (1 _ : FileHandler fn) → L io Unit
```

```
readLine : LinearIO io ⇒ (1 _ : FileHandler fn) →  
    L io {use=1} $ LPair' String $ FileHandler fn
```

```
f : LinearIO io ⇒ L io $ Maybe Bool
```

```
f = withOpenFile "foo" success (putStrLn "alas" *> pure Nothing)
```

```
where
```

```
    success : (1 _ : FileHandler _) → L io $ Maybe Bool
```

```
    success fh = do (str # fh) ← readLine fh  
                   closeFile fh  
                   pure $ Just (str == "x")
```

Использование ресурса: пример с файлами и IO

```
data FileHandler : FileName → Type where ...

interface (Monad io, LinearBind io) ⇒ FilesAPI io where
  withOpenFile : (fn : FileName) →
    (success : (1 _ : FileHandler fn) → L io a) →
    (fail : L io a) → L io a
  closeFile : (1 _ : FileHandler fn) → L io Unit
  readLine : (1 _ : FileHandler fn) →
    L io {use=1} $ LPair' String $ FileHandler fn

f : (FilesAPI io, HasLinearIO io) ⇒ L io $ Maybe Bool
f = withOpenFile "foo" success (putStrLn "alas" *> pure Nothing)
where
  success : (1 _ : FileHandler _) → L io $ Maybe Bool
  success fh = do (str # fh) ← readLine fh
    closeFile fh
    pure $ Just (str == "x")
```


Напоминание: возможности линейных типов

- Отражать в сигнатурах больше
- Отражать в сигнатурах новое
 - ▶ Ресурсы
 - ▶ Протоколы
- Оптимизация выполнения

Напоминание: возможности линейных типов

- Отражать в сигнатурах больше
- Отражать в сигнатурах новое
 - ▶ Ресурсы
 - ▶ **Протоколы**
- Оптимизация выполнения

Протоколы в типе

- Существуют session types

Протоколы в типе

- Существуют session types
 - ▶ Зависимые + линейные обеспечивают ту же выразительность

- Существуют session types
 - ▶ Зависимые + линейные обеспечивают ту же выразительность
 - ▶ Global session types vs. local session types

- Существуют session types
 - ▶ Зависимые + линейные обеспечивают ту же выразительность
 - ▶ Global session types vs. local session types
 - ▶ Binary session types vs. multiparty session types

- Существуют session types
 - ▶ Зависимые + линейные обеспечивают ту же выразительность
 - ▶ Global session types vs. **local session types**
 - ▶ **Binary session types** vs. multiparty session types

Протоколы в типе: логинистый пример

- Общение организовано сессиями

Протоколы в типе: логинистый пример

- Общение организовано сессиями
- За сессию можно (попытаться) залогиниться один раз

Протоколы в типе: логинистый пример

- Общение организовано сессиями
- За сессию можно (попытаться) залогиниться один раз
- При попытке залогинивания происходит проверка переданного ключа

Протоколы в типе: логинистый пример

- Общение организовано сессиями
- За сессию можно (попытаться) залогиниться один раз
- При попытке залогинивания происходит проверка переданного ключа
- Успешно залогинившись, обязательно вылогиниться

Протоколы в типе: логинистый пример

- Общение организовано сессиями
- За сессию можно (попытаться) залогиниться один раз
- При попытке залогинивания происходит проверка переданного ключа
- Успешно залогинившись, обязательно вылогиниться
- Залогинившись нужно зачекиниться
 - ▶ строго один раз
 - ▶ в произвольный момент пока залогиненный

Протоколы в типе: логинистый пример

- Общение организовано сессиями
- За сессию можно (попытаться) залогиниться один раз
- При попытке залогинивания происходит проверка переданного ключа
- Успешно залогинившись, обязательно вылогиниться
- Залогинившись нужно зачекиниться
 - ▶ строго один раз
 - ▶ в произвольный момент пока залогиненный
- Залогинившись можно поменять ключ

Протоколы в типе: логинистый пример

- Общение организовано сессиями
- За сессию можно (попытаться) залогиниться один раз
- При попытке залогинивания происходит проверка переданного ключа
- Успешно залогинившись, обязательно вылогиниться
- Залогинившись нужно зачекиниться
 - ▶ строго один раз
 - ▶ в произвольный момент пока залогиненный
- Залогинившись можно поменять ключ
- Залогинившись можно прочитать секретную строчку

Протоколы в типе: логинистый пример

```
data JournalState = NotYetCheckedIn | CheckedIn
data LoginState = Initial | LoggedIn JournalState | LoggedOut
prefix 9 @
data (@) : LoginState → Type where ...
```

Протоколы в типе: логинистый пример

```
data JournalState = NotYetCheckedIn | CheckedIn
data LoginState = Initial | LoggedIn JournalState | LoggedOut
prefix 9 @
data (@) : LoginState → Type where ...

interface (Monad m, LinearBind m) ⇒ SimpleProtocol m where
  beginSession : (1 _ : (1 _ : @ Initial) → L m a) → L m a
  endSession   : (1 _ : @ LoggedOut) → L m Unit
```


Протоколы в типе: логинистый пример

```
data JournalState = NotYetCheckedIn | CheckedIn
data LoginState = Initial | LoggedIn JournalState | LoggedOut
prefix 9 @
data (@) : LoginState → Type where ...

interface (Monad m, LinearBind m) ⇒ SimpleProtocol m where
  beginSession : (1 _ : (1 _ : @ Initial) → L m a) → L m a
  endSession   : (1 _ : @ LoggedOut) → L m Unit
  login        : (1 _ : @ Initial) → (name : String) → (key : Key) →
    L m {use=1} $ Res Bool \case
    True  ⇒ @ LoggedIn NotYetCheckedIn
    False ⇒ LPair' FailureReason (@ LoggedOut)
```

Протоколы в типе: логинистый пример

```
data JournalState = NotYetCheckedIn | CheckedIn
data LoginState = Initial | LoggedIn JournalState | LoggedOut
prefix 9 @
data (@) : LoginState → Type where ...

interface (Monad m, LinearBind m) ⇒ SimpleProtocol m where
  beginSession : (1 _ : (1 _ : @ Initial) → L m a) → L m a
  endSession   : (1 _ : @ LoggedOut) → L m Unit
  login        : (1 _ : @ Initial) → (name : String) → (key : Key) →
                 L m {use=1} $ Res Bool \case
                 True  ⇒ @ LoggedIn NotYetCheckedIn
                 False ⇒ LPair' FailureReason (@ LoggedOut)
```

```
data Res : (a : Type) → (a → Type) → Type where
  (#) : (val : a) → (1 r : t val) → Res a t
```

Протоколы в типе: логинистый пример

```
data JournalState = NotYetCheckedIn | CheckedIn
data LoginState = Initial | LoggedIn JournalState | LoggedOut
prefix 9 @
data (@) : LoginState → Type where ...

interface (Monad m, LinearBind m) ⇒ SimpleProtocol m where
  beginSession : (1 _ : (1 _ : @ Initial) → L m a) → L m a
  endSession   : (1 _ : @ LoggedOut) → L m Unit
  login        : (1 _ : @ Initial) → (name : String) → (key : Key) →
    L m {use=1} $ Res Bool \case
    True  ⇒ @ LoggedIn NotYetCheckedIn
    False ⇒ LPair' FailureReason (@ LoggedOut)
```

Протоколы в типе: логинистый пример

```
data JournalState = NotYetCheckedIn | CheckedIn
data LoginState = Initial | LoggedIn JournalState | LoggedOut
prefix 9 @
data (@) : LoginState → Type where ...

interface (Monad m, LinearBind m) ⇒ SimpleProtocol m where
  beginSession : (1 _ : (1 _ : @ Initial) → L m a) → L m a
  endSession   : (1 _ : @ LoggedOut) → L m Unit
  login        : (1 _ : @ Initial) → (name : String) → (key : Key) →
                 L m {use=1} $ Res Bool \case
                 True  ⇒ @ LoggedIn NotYetCheckedIn
                 False ⇒ LPair' FailureReason (@ LoggedOut)
  logout       : (1 _ : @ LoggedIn CheckedIn) → L m {use=1} (@ LoggedOut)
```

Протоколы в типе: логинистый пример

```
data JournalState = NotYetCheckedIn | CheckedIn
data LoginState = Initial | LoggedIn JournalState | LoggedOut
prefix 9 @
data (@) : LoginState → Type where ...

interface (Monad m, LinearBind m) ⇒ SimpleProtocol m where
  beginSession : (1 _ : (1 _ : @ Initial) → L m a) → L m a
  endSession   : (1 _ : @ LoggedOut) → L m Unit
  login        : (1 _ : @ Initial) → (name : String) → (key : Key) →
    L m {use=1} $ Res Bool \case
      True  ⇒ @ LoggedIn NotYetCheckedIn
      False ⇒ LPair' FailureReason (@ LoggedOut)
  logout       : (1 _ : @ LoggedIn CheckedIn) → L m {use=1} (@ LoggedOut)
  updateKey    : (1 _ : @ LoggedIn x) → (newKey : Key) →
    L m {use=1} $ LPair' (Maybe FailureReason) (@ LoggedIn x)
```

Протоколы в типе: логинистый пример

```
data JournalState = NotYetCheckedIn | CheckedIn
data LoginState = Initial | LoggedIn JournalState | LoggedOut
prefix 9 @
data (@) : LoginState → Type where ...

interface (Monad m, LinearBind m) ⇒ SimpleProtocol m where
  beginSession : (1 _ : (1 _ : @ Initial) → L m a) → L m a
  endSession   : (1 _ : @ LoggedOut) → L m Unit
  login        : (1 _ : @ Initial) → (name : String) → (key : Key) →
    L m {use=1} $ Res Bool \case
      True  ⇒ @ LoggedIn NotYetCheckedIn
      False ⇒ LPair' FailureReason (@ LoggedOut)
  logout       : (1 _ : @ LoggedIn CheckedIn) → L m {use=1} (@ LoggedOut)
  updateKey    : (1 _ : @ LoggedIn x) → (newKey : Key) →
    L m {use=1} $ LPair' (Maybe FailureReason) (@ LoggedIn x)
  readSecret   : (1 _ : @ LoggedIn x) →
    L m {use=1} $ LPair' String (@ LoggedIn x)
```

Протоколы в типе: логинистый пример

```
data JournalState = NotYetCheckedIn | CheckedIn
data LoginState = Initial | LoggedIn JournalState | LoggedOut
prefix 9 @
data (@) : LoginState → Type where ...

interface (Monad m, LinearBind m) ⇒ SimpleProtocol m where
  beginSession : (1 _ : (1 _ : @ Initial) → L m a) → L m a
  endSession   : (1 _ : @ LoggedOut) → L m Unit
  login        : (1 _ : @ Initial) → (name : String) → (key : Key) →
    L m {use=1} $ Res Bool \case
      True  ⇒ @ LoggedIn NotYetCheckedIn
      False ⇒ LPair' FailureReason (@ LoggedOut)
  logout       : (1 _ : @ LoggedIn CheckedIn) → L m {use=1} (@ LoggedOut)
  updateKey    : (1 _ : @ LoggedIn x) → (newKey : Key) →
    L m {use=1} $ LPair' (Maybe FailureReason) (@ LoggedIn x)
  readSecret   : (1 _ : @ LoggedIn x) →
    L m {use=1} $ LPair' String (@ LoggedIn x)
  checkIn     : (1 _ : @ LoggedIn NotYetCheckedIn) → (info : String) →
    L m {use=1} (@ LoggedIn CheckedIn)
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  ?foo
```


Протоколы в типе: логинистый пример

```
f : SimpleProtocol m ⇒ L m $ Either String Bool
f = beginSession \p ⇒ do
    ?foo
```

```
0 m : Type → Type
1 p : @ Initial
```

```
foo : L m (Either String Bool)
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  ans <- login p "Denis" denisKey
  ?foo
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  ans <- login p "Denis" denisKey
  ?foo
```

```
0 m : Type -> Type
0 p : @ Initial
1 ans : Res Bool (\{lcase:0} => if lcase
  then @ (LoggedIn NotYetCheckedIn)
  else LPair' FailureReason (@ LoggedOut))
```

```
foo : L m (Either String Bool)
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # s) <- login p "Denis" denisKey
  | (False # (reason # s)) => ?foo_bad
  ?foo_good
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m ⇒ L m $ Either String Bool
f = beginSession \p ⇒ do
  (True # s) ← login p "Denis" denisKey
  | (False # (reason # s)) ⇒ ?foo_bad
  ?foo_good
```

```
0 m : Type → Type
0 p : @ Initial
1 s : @ (LoggedIn NotYetCheckedIn)
-----
foo_good : L m (Either String Bool)
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m ⇒ L m $ Either String Bool
f = beginSession \p ⇒ do
  (True # s) ← login p "Denis" denisKey
  | (False # (reason # s)) ⇒ ?foo_bad
  ?foo_good
```

```
0 m : Type → Type
0 p : @ Initial
  reason : FailureReason
1 s : @ LoggedOut
-----
foo_bad : L m (Either String Bool)
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # s) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
?foo_good
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
?foo
```


Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
?foo
```

```
0 m : Type → Type
1 p : @ (LoggedIn NotYetCheckedIn)
-----
foo : L m (Either String Bool)
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
sec # p <- readSecret p
?foo
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
  sec # p <- readSecret p
  ?foo
```

```
0 m : Type -> Type
  sec : String
1 p : @ (LoggedIn NotYetCheckedIn)
-----
foo : L m (Either String Bool)
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
  sec # p <- readSecret p
  p <- logout p
  ?foo
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
  sec # p <- readSecret p
  p <- logout p
  ?foo
```

Error: While processing right hand side of f.
When unifying @ (LoggedIn NotYetCheckedIn) and @ (LoggedIn CheckedIn).
Mismatch between: NotYetCheckedIn and CheckedIn.

... /Example/Qt/One.idr:437:22--437:23

```
437 |           p <- logout p
      |                               ^
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
  sec # p <- readSecret p
  p <- checkIn p "read secret"
  p <- logout p
  ?foo
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
  sec # p <- readSecret p
  p <- checkIn p "read secret"
  p <- logout p
  pure $ Right $ sec = "foo"
```

Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
  sec # p <- readSecret p
  p <- checkIn p "read secret"
  p <- logout p
  pure $ Right $ sec = "foo"
```

Error: While processing right hand side of f.

There are 0 uses of linear name p.

Suggestion: linearly bounded variables must be used exactly once.

... /Example/Qt/One.idr:438:10--438:23

```
438 |           p <- logout p
      |           ^^^^^^^^^^^^^^^^^
```


Протоколы в типе: логинистый пример

```
f : SimpleProtocol m => L m $ Either String Bool
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
  sec # p <- readSecret p
  p <- checkIn p "read secret"
  p <- logout p
  endSession p
  pure $ Right $ sec = "foo"
```

Другие околопротокольные примеры

- Интерфейс агента игры, где, например, в зависимости от внешних воздействий множество возможных ответов меняется

Другие околопротокольные примеры

- Интерфейс агента игры, где, например, в зависимости от внешних воздействий множество возможных ответов меняется
- Жизненный цикл модуля операционной системы

Напоминание: возможности линейных типов

- Отражать в сигнатурах больше
- Отражать в сигнатурах новое
 - ▶ Ресурсы
 - ▶ Протоколы
- Оптимизация выполнения

Напоминание: возможности линейных типов

- Отражать в сигнатурах больше
- Отражать в сигнатурах новое
 - ▶ Ресурсы
 - ▶ Протоколы
- **Оптимизация выполнения**

Destructive update как оптимизация

```
f = runWithCreated MkParams \res =>  
  let (r # res) = depend res in  
  let () = destroy res in  
  MkUr r
```

Destructive update как оптимизация

```
f = runWithCreated MkParams \res =>
  let (r # res) = depend res in
  let () = destroy res in
  MkUr r
```

```
f = beginSession \p => do
  (True # p) <- login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
  sec # p <- readSecret p
  p <- checkIn p "read secret"
  p <- logout p
  endSession p
  pure $ Right $ sec = "foo"
```

Destructive update как оптимизация

```
f = runWithCreated MkParams \res =>
  let (r # res) = depend res in
  let () = destroy res in
  MkUr r
```

```
f = beginSession \p => do
  (True # p) ← login p "Denis" denisKey
  | (False # (reason # s)) => do
    endSession s
    pure $ Left $ show reason
  sec # p ← readSecret p
  p ← checkIn p "read secret"
  p ← logout p
  endSession p
  pure $ Right $ sec = "foo"
```

- $(r \# y) = f \ x$ и $(r \# y) \leftarrow f \ x$
МОГУТ КОМПИЛИРОВАТЬСЯ КАК ОБНОВЛЕНИЕ

Безопасные чистые mutable структуры данных

Безопасные чистые mutable структуры данных

Пример: монадический mutable массив

```
interface Monad m ⇒ MArray (0 ar : Nat → Type → Type) m where  
  new    : (n : Nat) → a → m (ar n a)  
  read   : Fin n → ar n a → m a  
  write  : Fin n → a → ar n a → m Unit
```

Безопасные чистые mutable структуры данных

Пример: монадический mutable массив

```
interface Monad m  $\Rightarrow$  MArray (0 ar : Nat  $\rightarrow$  Type  $\rightarrow$  Type) m where  
  new    : (n : Nat)  $\rightarrow$  a  $\rightarrow$  m (ar n a)  
  read   : Fin n  $\rightarrow$  ar n a  $\rightarrow$  m a  
  write  : Fin n  $\rightarrow$  a  $\rightarrow$  ar n a  $\rightarrow$  m Unit  
  
  freeze : ar n a  $\rightarrow$  m (IArray n a)    -- м.б. небезопасно --
```

Безопасные чистые mutable структуры данных

Пример: монадический mutable массив

```
interface Monad m  $\Rightarrow$  MArray (0 ar : Nat  $\rightarrow$  Type  $\rightarrow$  Type) m where
  new    : (n : Nat)  $\rightarrow$  a  $\rightarrow$  m (ar n a)
  read   : Fin n  $\rightarrow$  ar n a  $\rightarrow$  m a
  write  : Fin n  $\rightarrow$  a  $\rightarrow$  ar n a  $\rightarrow$  m Unit

  freeze : ar n a  $\rightarrow$  m (IArray n a)    -- м.б. небезопасно --

modify : MArray ar m  $\Rightarrow$  (a  $\rightarrow$  a)  $\rightarrow$  Fin n  $\rightarrow$  ar n a  $\rightarrow$  m Unit
modify f i arr = do
  x  $\leftarrow$  read i arr
  write i (f x) arr
```

Безопасные чистые mutable структуры данных

Пример: монадический mutable массив

```
interface Monad m  $\Rightarrow$  MArray (0 ar : Nat  $\rightarrow$  Type  $\rightarrow$  Type) m where
  new    : (n : Nat)  $\rightarrow$  a  $\rightarrow$  m (ar n a)
  read   : Fin n  $\rightarrow$  ar n a  $\rightarrow$  m a
  write  : Fin n  $\rightarrow$  a  $\rightarrow$  ar n a  $\rightarrow$  m Unit

  freeze : ar n a  $\rightarrow$  m (IArray n a)    -- м.б. небезопасно --

modify : MArray ar m  $\Rightarrow$  (a  $\rightarrow$  a)  $\rightarrow$  Fin n  $\rightarrow$  ar n a  $\rightarrow$  m Unit
modify f i arr = do
  x  $\leftarrow$  read i arr
  write i (f x) arr

f : MArray ar m  $\Rightarrow$  Fin n  $\rightarrow$  ar n Nat  $\rightarrow$  m Nat
f i arr = do original  $\leftarrow$  read i arr
           modify (+1) i arr
           pure original
```

Безопасные чистые mutable структуры данных

Линейный mutable массив

```
data LArray : Nat → Type → Type where [external]
```

Безопасные чистые mutable структуры данных

Линейный mutable массив

```
data LArray : Nat → Type → Type where [external]
```

```
withNew : (n : Nat) → a → (1 _ : (1 _ : LArray n a) → Ur b) → b
```

```
read    : Fin n → (1 _ : LArray n a) → LPair' a $ LArray n a
```

```
write   : Fin n → a → (1 _ : LArray n a) → LArray n a
```

Безопасные чистые mutable структуры данных

Линейный mutable массив

```
data LArray : Nat → Type → Type where [external]
```

```
withNew : (n : Nat) → a → (1 _ : (1 _ : LArray n a) → Ur b) → b
```

```
read    : Fin n → (1 _ : LArray n a) → LPair' a $ LArray n a
```

```
write   : Fin n → a → (1 _ : LArray n a) → LArray n a
```

```
freeze  : (1 _ : LArray n a) → Ur $ IArray n a
```


Безопасные чистые mutable структуры данных

Линейный mutable массив

```
data LArray : Nat → Type → Type where [external]
```

```
withNew : (n : Nat) → a → (1 _ : (1 _ : LArray n a) → Ur b) → b
```

```
read : Fin n → (1 _ : LArray n a) → LPair' a $ LArray n a
```

```
write : Fin n → a → (1 _ : LArray n a) → LArray n a
```

```
freeze : (1 _ : LArray n a) → Ur $ IArray n a
```

```
modify : (a → a) → Fin n → (1 _ : LArray n a) → LArray n a
```

```
modify f i arr =
```

```
  let x # arr = read i arr
```

```
  in write i (f x) arr
```

Безопасные чистые mutable структуры данных

Линейный mutable массив

```
data LArray : Nat → Type → Type where [external]
```

```
withNew : (n : Nat) → a → (1 _ : (1 _ : LArray n a) → Ur b) → b
```

```
read    : Fin n → (1 _ : LArray n a) → LPair' a $ LArray n a
```

```
write   : Fin n → a → (1 _ : LArray n a) → LArray n a
```

```
freeze  : (1 _ : LArray n a) → Ur $ IArray n a
```

```
modify  : (a → a) → Fin n → (1 _ : LArray n a) → LArray n a
```

```
modify f i arr =
```

```
  let x # arr = read i arr
```

```
  in write i (f x) arr
```

```
f : Fin n → (1 _ : LArray n Nat) → LPair' Nat $ LArray n Nat
```

```
f i arr = let original # arr = read i arr
```

```
          arr = modify (+1) i arr
```

```
          in original # arr
```

Когда?

Когда?

- В рантайме

Когда?

- В рантайме

- ▶ 0 — никогда

Когда?

- В рантайме

- ▶ 0 — *никогда*

- ▶ 1 — *ровно один раз*

Когда?

- В рантайме
 - ▶ 0 — *никогда*
 - ▶ 1 — *ровно один раз*
- При компиляции — неограниченно

Глава 4

«Эн»

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x

repN : Nat → (a → a) → (a → a)
repN Z     f = id
repN (S n) f = repN n (f . f)
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x

repN : Nat → (a → a) → (a → a)
repN Z     f = f
repN (S n) f = repN n (f . f)
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x

repN : Nat → (a → a) → (a → a)
repN Z     f = f
repN (S n) f = repN n f . f
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x

repN : Nat → (a → a) → (a → a)
repN Z     f = id
repN (S n) f = repN n f . f
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : Nat → (a → a) → (a → a)
repN Z     f = id
repN (S n) f = repN n f . f
```

Каждый вариант подходил под сигнатуру!

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : (n : Nat) → ([n] _ : a → a) → (a → a)
repN Z      f = id
repN (S n) f = repN n f . f
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : (n : Nat) → ([n] _ : a → a) → (a → a)
repN Z      f = id
repN (S n) f = repN n f . f
```

↑ *Не настоящий синтаксис Idris 2!*

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : (n : Nat) → ([n] _ : a → a) → (a → a)
repN Z      f = id
repN (S n) f = repN n f . f
```

↑ *Не настоящий синтаксис Idris 2!*

```
foldr : (op : a → b → b) →
        (init : b) →
        Vect n a → b
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : (n : Nat) → ([n] _ : a → a) → (a → a)
repN Z      f = id
repN (S n) f = repN n f . f
```

↑ *Не настоящий синтаксис Idris 2!* ↓

```
foldr : ([n] op : a → b → b) →
        (init : b) →
        Vect n a → b
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : (n : Nat) → ([n] _ : a → a) → (a → a)
repN Z      f = id
repN (S n) f = repN n f . f
```

↑ *Не настоящий синтаксис Idris 2!* ↓

```
foldr : ([n] op : a → ([k] _ : b) → b) →
        (init : b) →
        Vect n a → b
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : (n : Nat) → ([n] _ : a → a) → (a → a)
repN Z      f = id
repN (S n) f = repN n f . f
```

↑ *Не настоящий синтаксис Idris 2!* ↓

```
foldr : ([n] op : a → ([k] _ : b) → b) →
        (? init : b) →
        Vect n a → b
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : (n : Nat) → ([n] _ : a → a) → (a → a)
repN Z      f = id
repN (S n) f = repN n f . f
```

↑ *Не настоящий синтаксис Idris 2!* ↓

```
foldr : ([n] op : a → ([k] _ : b) → b) →
        ([power k n] init : b) →
        Vect n a → b
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : (n : Nat) → ([n] _ : a → a) → (a → a)
repN Z      f = id
repN (S n) f = repN n f . f
```

↑ *Не настоящий синтаксис Idris 2!* ↓

```
foldr : ([n] op : a → ([k] _ : b) → b) →
        ([power k n] init : b) →      -- Меру надо знать, конечно --
        Vect n a → b
```

n раз

```
-- repN 0 f x ~ x
-- repN 1 f x ~ f x
-- repN 2 f x ~ f . f $ x
-- repN 3 f x ~ f . f . f $ x
```

```
repN : (n : Nat) → ([n] _ : a → a) → (a → a)
repN Z      f = id
repN (S n) f = repN n f . f
```

↑ *Не настоящий синтаксис Idris 2!* ↓

```
foldr : ([n] op : a → ([k] _ : b) → b) →
        ([power k n] init : b) →      -- Меру надо знать, конечно --
        Vect n a → b
```

```
map : ([n] _ : a → b) → Vect n a → Vect n b
```

Глава 5

От и до

Точно?

```
data Maybe a = Just a | Nothing
isJust : (1 _ : Maybe a) → Bool
```

Точно?

```
data Maybe a = Just a | Nothing
```

```
isJust : (1 _ : Maybe a) → Bool
```

```
fromMaybe : a → (1 _ : Maybe a) → a
```

Точно?

```
data Maybe a = Just a | Nothing
```

```
isJust : (1 _ : Maybe a) → Bool
```

```
fromMaybe : a → (1 _ : Maybe a) → a
```

```
fromMaybe : ([0..1] _ : a) → (1 _ : Maybe a) → a
```

Точно?

```
data Maybe a = Just a | Nothing
```

```
isJust : (1 _ : Maybe a) → Bool
```

```
fromMaybe : a → (1 _ : Maybe a) → a
```

```
fromMaybe : ([0..1] _ : a) → (1 _ : Maybe a) → a
```

```
fromMaybe : (1 m : Maybe a) →  
  ([if isJust m then 1 else 0] _ : a) → a
```

Точно?

```
data Maybe a = Just a | Nothing
```

```
isJust : (1 _ : Maybe a) → Bool
```

```
fromMaybe : a → (1 _ : Maybe a) → a
```

```
fromMaybe : ([0..1] _ : a) → (1 _ : Maybe a) → a
```

```
fromMaybe : (1 m : Maybe a) →  
              ([if isJust m then 1 else 0] _ : a) → a
```

Упомянутые ранее частные случаи

- [0..1] — аффинные
- [1..] — relevant

Неточные комбинации

`fromMaybe` : (`[0..1]` `_` : `a`) \rightarrow (`1` `_` : `Maybe a`) \rightarrow `a`

`map` : (`[n]` `_` : `a` \rightarrow `b`) \rightarrow `Vect n a` \rightarrow `Vect n b`

Неточные комбинации

`fromMaybe` : (`[0..1]` `_` : `a`) \rightarrow (`1` `_` : `Maybe a`) \rightarrow `a`

`map` : (`[n]` `_` : `a` \rightarrow `b`) \rightarrow `Vect n a` \rightarrow `Vect n b`

`f` : `a` \rightarrow `Vect n (Maybe a)` \rightarrow `Vect n a`

`f d = map (fromMaybe d)`

Неточные комбинации

`fromMaybe` : (`[0..1]` `_` : `a`) \rightarrow (`1` `_` : `Maybe a`) \rightarrow `a`

`map` : (`[n]` `_` : `a` \rightarrow `b`) \rightarrow `Vect n a` \rightarrow `Vect n b`

`f` : `a` \rightarrow `Vect n (Maybe a)` \rightarrow `Vect n a`

`f d = map (fromMaybe d)`

Сколько раз используется d?

Неточные комбинации

`fromMaybe` : (`[0..1]` `_` : `a`) \rightarrow (`1` `_` : `Maybe a`) \rightarrow `a`

`map` : (`[n]` `_` : `a` \rightarrow `b`) \rightarrow `Vect n a` \rightarrow `Vect n b`

`f` : (`[0..n]` `_` : `a`) \rightarrow `Vect n (Maybe a)` \rightarrow `Vect n a`
`f d = map (fromMaybe d)`

Глава 6

Полукольца

(Полу)кольца всевластия?



$$f : ([n] _ : A) \rightarrow B$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

$$fg : (? _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

- $h : ([m] _ : B) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

- $h : ([m] _ : B) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

$$fh : (? _ : A) \rightarrow C$$

$$fh \ a = h \ (f \ a)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

- $h : ([m] _ : B) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h (f \ a)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

- $h : ([m] _ : B) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h (f \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

- $h : ([m] _ : B) \rightarrow C$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h (f \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$hj : (? _ : B) \rightarrow (C, C)$$

$$hj \ b = (h \ b, j \ b)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

- $h : ([m] _ : B) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h (f \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$hj : ([m] _ : B) \rightarrow (C, C)$$

$$hj \ b = (h \ b, j \ b)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

- $h : ([m] _ : B) \rightarrow C$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h \ (f \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$hj : ([m] _ : B) \rightarrow (C, C)$$

$$hj \ b = (h \ b, j \ b)$$

$$fj : (? _ : A) \rightarrow C$$

$$fj \ a = j \ (f \ a)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

- $h : ([m] _ : B) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h (f \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$hj : ([m] _ : B) \rightarrow (C, C)$$

$$hj \ b = (h \ b, j \ b)$$

$$fj : (0 _ : A) \rightarrow C$$

$$fj \ a = j (f \ a)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

- $h : ([m] _ : B) \rightarrow C$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h \ (f \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$hj : ([m] _ : B) \rightarrow (C, C)$$

$$hj \ b = (h \ b, j \ b)$$

$$fj : (0 _ : A) \rightarrow C$$

$$fj \ a = j \ (f \ a)$$

- $i : (1 _ : B) \rightarrow C$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$hj : ([m] _ : B) \rightarrow (C, C)$$

$$hj \ b = (h \ b, j \ b)$$

$$fj : (0 _ : A) \rightarrow C$$

$$fj \ a = j \ (f \ a)$$

- $h : ([m] _ : B) \rightarrow C$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h \ (f \ a)$$

- $i : (1 _ : B) \rightarrow C$

$$hi : (? _ : B) \rightarrow (C, C)$$

$$hi \ b = (h \ b, i \ b)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$hj : ([m] _ : B) \rightarrow (C, C)$$

$$hj \ b = (h \ b, j \ b)$$

$$fj : (0 _ : A) \rightarrow C$$

$$fj \ a = j \ (f \ a)$$

- $h : ([m] _ : B) \rightarrow C$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h \ (f \ a)$$

- $i : (1 _ : B) \rightarrow C$

$$hi : ([m + 1] _ : B) \rightarrow (C, C)$$

$$hi \ b = (h \ b, i \ b)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$hj : ([m] _ : B) \rightarrow (C, C)$$

$$hj \ b = (h \ b, j \ b)$$

$$fj : (0 _ : A) \rightarrow C$$

$$fj \ a = j \ (f \ a)$$

- $h : ([m] _ : B) \rightarrow C$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h \ (f \ a)$$

- $i : (1 _ : B) \rightarrow C$

$$hi : ([m + 1] _ : B) \rightarrow (C, C)$$

$$hi \ b = (h \ b, i \ b)$$

$$fi : (? _ : A) \rightarrow C$$

$$fi \ a = i \ (f \ a)$$

$$f : ([n] _ : A) \rightarrow B$$

- $g : ([m] _ : A) \rightarrow C$

$$fg : ([n + m] _ : A) \rightarrow (B, C)$$

$$fg \ a = (f \ a, g \ a)$$

- $j : (0 _ : B) \rightarrow C$

$$hj : ([m] _ : B) \rightarrow (C, C)$$

$$hj \ b = (h \ b, j \ b)$$

$$fj : (0 _ : A) \rightarrow C$$

$$fj \ a = j \ (f \ a)$$

- $h : ([m] _ : B) \rightarrow C$

$$fh : ([n * m] _ : A) \rightarrow C$$

$$fh \ a = h \ (f \ a)$$

- $i : (1 _ : B) \rightarrow C$

$$hi : ([m + 1] _ : B) \rightarrow (C, C)$$

$$hi \ b = (h \ b, i \ b)$$

$$fi : ([n] _ : A) \rightarrow C$$

$$fi \ a = i \ (f \ a)$$

Полукольца

Полукольца

- Сложение

- ▶ Коммутативно: $a + b = b + a$
- ▶ Ассоциативно: $(a + b) + c = a + (b + c)$
- ▶ Существует ноль: $a + 0 = 0 + a = a$

Полукольца

- Сложение

- ▶ Коммутативно: $a + b = b + a$
- ▶ Ассоциативно: $(a + b) + c = a + (b + c)$
- ▶ Существует ноль: $a + 0 = 0 + a = a$

- Умножение

- ▶ Ассоциативно: $(a * b) * c = a * (b * c)$
- ▶ Ноль от сложения даёт ноль: $a * 0 = 0 * a = 0$
- ▶ Коммутативность **не** требуется

Полукольца

- Сложение

- ▶ Коммутативно: $a + b = b + a$
- ▶ Ассоциативно: $(a + b) + c = a + (b + c)$
- ▶ Существует ноль: $a + 0 = 0 + a = a$

- Умножение

- ▶ Ассоциативно: $(a * b) * c = a * (b * c)$
- ▶ Ноль от сложения даёт ноль: $a * 0 = 0 * a = 0$
- ▶ Коммутативность **не** требуется

- Дистрибутивность сложения и умножения

- ▶ Левая $a * (b + c) = a * b + a * c$
- ▶ Правая $(a + b) * c = a * c + b * c$

Полукольца с предпорядком

- Сложение

- ▶ Коммутативно: $a + b = b + a$
- ▶ Ассоциативно: $(a + b) + c = a + (b + c)$
- ▶ Существует ноль: $a + 0 = 0 + a = a$

- Умножение

- ▶ Ассоциативно: $(a * b) * c = a * (b * c)$
- ▶ Ноль от сложения даёт ноль: $a * 0 = 0 * a = 0$
- ▶ Коммутативность **не** требуется

- Дистрибутивность сложения и умножения

- ▶ Левая $a * (b + c) = a * b + a * c$
- ▶ Правая $(a + b) * c = a * c + b * c$

- Предпорядок

- ▶ Рефлексивность: $a \prec a$
- ▶ Транзитивность: $a \prec b \wedge b \prec c \Rightarrow a \prec c$

Полукольца с предпорядком

Полукольца с предпорядком

- $\{0, 1, \omega\}$

Полукольца с предпорядком

- $\{0, 1, \omega\}$
- Натуральные числа, \equiv

Полукольца с предпорядком

- $\{0, 1, \omega\}$
- Натуральные числа, \equiv
- Натуральные числа, \leq

Полукольца с предпорядком

- $\{0, 1, \omega\}$
- Натуральные числа, \equiv
- Натуральные числа, \leq
- Натуральные числа с ω

Полукольца с предпорядком

- $\{0, 1, \omega\}$
- Натуральные числа, \equiv
- Натуральные числа, \leq
- Натуральные числа с ω
- Отрезок натуральных чисел

Полукольца с предпорядком

- $\{0, 1, \omega\}$
- Натуральные числа, \equiv
- Натуральные числа, \leq
- Натуральные числа с ω
- Отрезок натуральных чисел
- Пара полуколец с предпорядком

Полукольца с предпорядком

- $\{0, 1, \omega\}$
- Натуральные числа, \equiv
- Натуральные числа, \leq
- Натуральные числа с ω
- Отрезок натуральных чисел
- Пара полуколец с предпорядком
- Приватность

Полукольца с предпорядком

- $\{0, 1, \omega\}$
- Натуральные числа, \equiv
- Натуральные числа, \leq
- Натуральные числа с ω
- Отрезок натуральных чисел
- Пара полуколец с предпорядком
- Приватность
- ...

Полукольцо приватности

```
record User where
  constructor MkUser
  [Public]  nick  : String
  [Private] email : String
```

Полукольцо приватности

```
record User where
  constructor MkUser
  [Public]  nick  : String
  [Private] email : String

data Pu : Type → Type where
  MkPu : ([Public] _ : a) → Pu a

data Pr : Type → Type where
  MkPr : ([Private] _ : a) → Pr a
```

Полукольцо приватности

```
record User where
  constructor MkUser
  [Public]  nick  : String
  [Private] email : String

data Pu : Type → Type where
  MkPu : ([Public] _ : a) → Pu a

data Pr : Type → Type where
  MkPr : ([Private] _ : a) → Pr a

f : Fin n → Vect n User → Pu String
f i v = let MkUser {email=str, ...} = index i v
        in str
```

Полукольцо приватности

```
record User where
  constructor MkUser
  [Public]  nick  : String
  [Private] email : String

data Pu : Type → Type where
  MkPu : ([Public] _ : a) → Pu a

data Pr : Type → Type where
  MkPr : ([Private] _ : a) → Pr a

f : Fin n → Vect n User → Pu String
f i v = let MkUser {nick=str, ...} = index i v
        in str
```

Глава 7

Впихнуть невпихуемое

При таких свиньях как-то сам становишься...

`f : Nat → A`

`g : (1 _ : A) → B`

-- много не просит

При таких свиньях как-то сам становишься...

```
f : Nat → A
```

```
g : (1 _ : A) → B
```

```
-- много не просит
```

```
let x = f 5 in ?foo
```


При таких свиньях как-то сам становишься...

```
f : Nat → A
```

```
g : (1 _ : A) → B
```

```
let x = f 5 in g x
```

```
-- много не просит
```

```
-- можем дать больше, чем просят
```

При таких свиньях как-то сам становишься...

```
map : (a → b) → List a → List b  -- ничего не просит от аргумента
```

При таких свиньях как-то сам становишься...

```
map : (a → b) → List a → List b  -- ничего не просит от аргумента
inc  : (1 _ : Nat) → Nat           -- функция кое-что обещает
```

При таких свиньях как-то сам становишься...

```
map : (a → b) → List a → List b -- ничего не просит от аргумента
inc  : (1 _ : Nat) → Nat          -- функция кое-что обещает
mi   : List Nat → List Nat
mi = map inc                       -- дали больше, чем просят
```

При таких свиньях как-то сам становишься...

```
map : (a → b) → List a → List b -- ничего не просит от аргумента
inc : (1 _ : Nat) → Nat           -- функция кое-что обещает
mi : List Nat → List Nat
mi = map inc                       -- дали больше, чем просят
```

```
interface Con w where
  constructor MkCon
  con : Nat → w                    -- con : Con w ⇒ Nat → w
```

При таких свиньях как-то сам становишься...

```
map : (a → b) → List a → List b -- ничего не просит от аргумента
inc : (1 _ : Nat) → Nat           -- функция кое-что обещает
mi  : List Nat → List Nat
mi = map inc                       -- дали больше, чем просят
```

```
interface Con w where
  constructor MkCon
  con : Nat → w -- con : Con w ⇒ Nat → w
mkCon : (a → b) → Con (a → b) -- instance первого порядка
mkCon f = MkCon {con = \_ ⇒ f}
```

При таких свиньях как-то сам становишься...

```
map : (a → b) → List a → List b -- ничего не просит от аргумента
inc : (1 _ : Nat) → Nat           -- функция кое-что обещает
mi  : List Nat → List Nat
mi = map inc                       -- дали больше, чем просят
```

```
interface Con w where
  constructor MkCon
  con : Nat → w -- con : Con w ⇒ Nat → w
mkCon : (a → b) → Con (a → b) -- instance первого порядка
mkCon f = MkCon {con = \_ ⇒ f}
bang  : (a → b) → (1 _ : a) → b
```

При таких свиньях как-то сам становишься...

```
map : (a → b) → List a → List b -- ничего не просит от аргумента
inc : (1 _ : Nat) → Nat           -- функция кое-что обещает
mi  : List Nat → List Nat
mi = map inc                       -- дали больше, чем просят
```

```
interface Con w where
  constructor MkCon
  con : Nat → w -- con : Con w ⇒ Nat → w

mkCon : (a → b) → Con (a → b) -- instance первого порядка
mkCon f = MkCon {con = \_ ⇒ f}

bang : (a → b) → (1 _ : a) → b
bang f = let cc = mkCon f in ?foo
```


При таких свиньях как-то сам становишься...

```
map : (a → b) → List a → List b -- ничего не просит от аргумента
inc : (1 _ : Nat) → Nat           -- функция кое-что обещает
mi : List Nat → List Nat
mi = map inc                       -- дали больше, чем просят
```

```
interface Con w where
  constructor MkCon
  con : Nat → w -- con : Con w ⇒ Nat → w
mkCon : (a → b) → Con (a → b) -- instance первого порядка
mkCon f = MkCon {con = \_ ⇒ f}
bang : (a → b) → (1 _ : a) → b
bang f = let cc = mkCon f in ?foo -- cc : Con (a → b)
```

При таких свиньях как-то сам становишься...

```
map : (a → b) → List a → List b -- ничего не просит от аргумента
inc : (1 _ : Nat) → Nat           -- функция кое-что обещает
mi  : List Nat → List Nat
mi = map inc                       -- дали больше, чем просят
```

```
interface Con w where
```

```
  constructor MkCon
```

```
  con : Nat → w
```

```
-- con : Con w ⇒ Nat → w
```

```
mkCon : (a → b) → Con (a → b)
```

```
-- instance первого порядка
```

```
mkCon f = MkCon {con = \_ ⇒ f}
```

```
bang : (a → b) → (1 _ : a) → b
```

```
bang f = let cc = mkCon f in con 5 -- утекло :-)
```

Какая сложна

- Не только функции контравариантны по своим аргументам

Какая сложна

- Не только функции контравариантны по своим аргументам
- Полиморфизм

Какая сложна

- Не только функции контравариантны по своим аргументам
- Полиморфизм
 - ▶ Рекурсивные структуры данных?

Какая сложна

- Не только функции контравариантны по своим аргументам
- Полиморфизм
 - ▶ Рекурсивные структуры данных?
- Дерайвинг

Какая сложна

- Не только функции контравариантны по своим аргументам
- Полиморфизм
 - ▶ Рекурсивные структуры данных?
- Дерайвинг
- ...

Глава 8

Спецификации спецификаций

QTT \Rightarrow GrTT

- Разрабатывается в Granule Project

QTT \Rightarrow GrTT

- Разрабатывается в Granule Project
- Некоторые отличия

QTT \Rightarrow GrTT

- Разрабатывается в Granule Project
- Некоторые отличия
 - ▶ Quantity привязана к типу

QTT \implies GrTT

- Разрабатывается в Granule Project
- Некоторые отличия
 - ▶ Quantity привязана к типу

`new` : (n : Nat) \rightarrow a \rightarrow [1] LArray n a

QTT \implies GrTT

- Разрабатывается в Granule Project
- Некоторые отличия
 - ▶ Quantity привязана к типу

```
new   : (n : Nat) → a → [1] LArray n a
read  : Fin n → [1] LArray n a → (a, [1] LArray n a)
```

QTT \implies GrTT

- Разрабатывается в Granule Project
- Некоторые отличия
 - ▶ Quantity привязана к типу

```
new   : (n : Nat) → a → [1] LArray n a
read  : Fin n → [1] LArray n a → (a, [1] LArray n a)
write : Fin n → a → [1] LArray n a → [1] LArray n a
```

QTT \implies GrTT

- Разрабатывается в Granule Project
- Некоторые отличия
 - ▶ Quantity привязана к типу

```
new    : (n : Nat) → a → [1] LArray n a
read   : Fin n → [1] LArray n a → (a, [1] LArray n a)
write  : Fin n → a → [1] LArray n a → [1] LArray n a
freeze : [1] LArray n a → [] Vect n a
```

QTT \implies GrTT

- Разрабатывается в Granule Project
- Некоторые отличия

- ▶ Quantity привязана к типу

```
new   : (n : Nat) → a → [1] LArray n a
read  : Fin n → [1] LArray n a → (a, [1] LArray n a)
write : Fin n → a → [1] LArray n a → [1] LArray n a
freeze : [1] LArray n a → [] Vect n a
```

- ▶ Quantity'ей две: одна рантаймовая, другая спецификационная

```
id : {a : [0, 2] Type} → [k, 0] a → [k, 0] a
```


Глава 9

Заключение

Блеск

- Более точные намерения, известные компилятору

Блеск

- Более точные намерения, известные компилятору
 - ▶ проверка при компиляции

Блеск

- Более точные намерения, известные компилятору
 - ▶ проверка при компиляции
 - ▶ более эффективное исполнение

Блеск

- Более точные намерения, известные компилятору
 - ▶ проверка при компиляции
 - ▶ более эффективное исполнение
- Исправление фундаментального косяка с parametricity

Блеск

- Более точные намерения, известные компилятору
 - ▶ проверка при компиляции
 - ▶ более эффективное исполнение
- Исправление фундаментального косяка с parametricity
- Новые возможности: например, протоколы в типе

Блеск

- Более точные намерения, известные компилятору
 - ▶ проверка при компиляции
 - ▶ более эффективное исполнение
- Исправление фундаментального косяка с parametricity
- Новые возможности: например, протоколы в типе
- Возможности более безопасных интерфейсов

Блеск и нищета

- Более точные намерения, известные компилятору
 - ▶ проверка при компиляции
 - ▶ более эффективное исполнение
- Исправление фундаментального косяка с parametricity
- Новые возможности: например, протоколы в типе
- Возможности более безопасных интерфейсов

- Много нерешённых проблем

Блеск и нищета

- Более точные намерения, известные компилятору
 - ▶ проверка при компиляции
 - ▶ более эффективное исполнение
- Исправление фундаментального косяка с parametricity
- Новые возможности: например, протоколы в типе
- Возможности более безопасных интерфейсов

- Много нерешённых проблем
- Много вариативности: ещё непонятно, как хорошо, а как нет

Блеск и нищета

- Более точные намерения, известные компилятору
 - ▶ проверка при компиляции
 - ▶ более эффективное исполнение
- Исправление фундаментального косяка с parametricity
- Новые возможности: например, протоколы в типе
- Возможности более безопасных интерфейсов

- Много нерешённых проблем
- Много вариативности: ещё непонятно, как хорошо, а как нет
- Вопросы удобства

Блеск и нищета

- Более точные намерения, известные компилятору
 - ▶ проверка при компиляции
 - ▶ более эффективное исполнение
- Исправление фундаментального косяка с parametricity
- Новые возможности: например, протоколы в типе
- Возможности более безопасных интерфейсов

- Много нерешённых проблем
- Много вариативности: ещё непонятно, как хорошо, а как нет
- Вопросы удобства

Поэтому, область развивается с безумной скоростью прямо сейчас

Сладкий мир циферек в типах

- Бывают и другие циферьки (и не только циферьки), например

Сладкий мир циферек в типах

- Бывают и другие циферки (и не только циферки), например
 - ▶ номер вселенной в теории зависимых типов

Сладкий мир циферек в типах

- Бывают и другие циферки (и не только циферки), например
 - ▶ номер вселенной в теории зависимых типов
 - ▶ гомотопический уровень в гомотопической теории типов

Сладкий мир циферек в типах

- Бывают и другие циферьки (и не только циферьки), например
 - ▶ номер вселенной в теории зависимых типов
 - ▶ гомотопический уровень в гомотопической теории типов
 - ▶ мера ленивости вычисления

Сладкий мир циферек в типах

- Бывают и другие циферьки (и не только циферьки), например
 - ▶ номер вселенной в теории зависимых типов
 - ▶ гомотопический уровень в гомотопической теории типов
 - ▶ мера ленивости вычисления
 - ▶ мера сложности вычисления значения этого типа

Сладкий мир циферек в типах

- Бывают и другие циферки (и не только циферки), например
 - ▶ номер вселенной в теории зависимых типов
 - ▶ гомотопический уровень в гомотопической теории типов
 - ▶ мера ленивости вычисления
 - ▶ мера сложности вычисления значения этого типа
 - ▶ мера количества потребления энергии для вычисления

Сладкий мир циферек в типах

- Бывают и другие циферки (и не только циферки), например
 - ▶ номер вселенной в теории зависимых типов
 - ▶ гомотопический уровень в гомотопической теории типов
 - ▶ мера ленивости вычисления
 - ▶ мера сложности вычисления значения этого типа
 - ▶ мера количества потребления энергии для вычисления
- Тут тоже бывает полиморфность

Сладкий мир циферек в типах

- Бывают и другие циферки (и не только циферки), например
 - ▶ номер вселенной в теории зависимых типов
 - ▶ гомотопический уровень в гомотопической теории типов
 - ▶ мера ленивости вычисления
 - ▶ мера сложности вычисления значения этого типа
 - ▶ мера количества потребления энергии для вычисления
- Тут тоже бывает полиморфность
 - ▶ причём, для первых двух она уже достигнута в некоторых языках

Сладкий мир циферек в типах

- Бывают и другие циферки (и не только циферки), например
 - ▶ номер вселенной в теории зависимых типов
 - ▶ гомотопический уровень в гомотопической теории типов
 - ▶ мера ленивости вычисления
 - ▶ мера сложности вычисления значения этого типа
 - ▶ мера количества потребления энергии для вычисления
- Тут тоже бывает полиморфность
 - ▶ причём, для первых двух она уже достигнута в некоторых языках
 - ▶ полиморфность по ленивости — прямо ах, как хотелось бы

Материалы

- Dominic Orchard.
Quantitative program reasoning in Granule via graded modal types (2019)
- Vilem-Benjamin Liepelt.
Quantitative program reasoning with graded modal types (2019)
- Dominic Orchard, Vilem-Benjamin Liepelt, Harley Eades III.
Quantitative Program Reasoning with Graded Modal Types (2019)
- Benjamin Moon. Towards Graded Modal Dependent Types (2020)
- Edwin Brady. Quantitative Types in Idris 2 (2020)
- Edwin Brady. Edwin Brady Tells Us What's New in Idris 2 (2020)
- Jan de Muijnck-Hughes, Edwin Brady, Wim Vanderbauwhede.
Value-dependent Session Design in a Dependently Typed Language (2019)
- Liam O'Connor. Refinement through Restraint: Bringing Down the Cost of Verification (2016)

Спасибо

Вопросы?