Зависимые типы + property-based testing = ❤️

Денис Буздало́в

19 марта 2024

О докладе

О докладе

- Зависимые типы

О докладе

- Зависимые типы
- property-based testing

О докладе

- Зависимые типы
- property-based testing
- +

О докладе

- Зависимые типы
- property-based testing
- +
- = ❤️

О докладе

- Зависимые типы
- property-based testing
- +
- = ❤️

- Будет много кода

О докладе

- Зависимые типы
- property-based testing
- +
- = ❤️

- Будет много кода, в основном на Idris

О докладе

- Зависимые типы
- property-based testing
- +
- = ❤️

- Будет много кода, в основном на Idris
- Синтаксис почти как Haskell

О докладе

- Зависимые типы
- property-based testing
- +
- = ❤️

- Будет много кода, в основном на Idris
- Синтаксис почти как Haskell, только лучше 😜

О докладе

- Зависимые типы
- property-based testing
- +
- = ❤️

- Будет много кода, в основном на Idris
- Синтаксис почти как Haskell, только лучше 😜
- Пробежимся только по верхам

О докладе

- Зависимые типы
- property-based testing
- +
- = ❤️

- Будет много кода, в основном на Idris
- Синтаксис почти как Haskell, только лучше 😜
- Пробежимся только по верхам, но всё равно будет сложно

О докладе

- Зависимые типы
- property-based testing
- +
- = ❤️

- Будет много кода, в основном на Idris
- Синтаксис почти как Haskell, только лучше 😜
- Пробежимся только по верхам, но всё равно будет сложно
- Сложность будет расти нелинейно

О докладе

- Зависимые типы
- property-based testing
- +
- = ❤️

- Будет много кода, в основном на Idris
- Синтаксис почти как Haskell, только лучше 😜
- Пробежимся только по верхам, но всё равно будет сложно
- Сложность будет расти нелинейно, задавайте вопросы по ходу

Чуточку о синтаксисе

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

Чуточку о синтаксисе

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

```
data Maybe : Type → Type where
  Nothing : Maybe a
  Just    : a → Maybe a
```

Чуточку о синтаксисе

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

```
data Maybe : Type → Type where
  Nothing : Maybe a
  Just    : a → Maybe a
```

```
length : List a → Nat
length []      = 0
length (_::xs) = 1 + length xs
```

Чуточку о синтаксисе

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

```
data Maybe : Type → Type where
  Nothing : Maybe a
  Just    : a → Maybe a
```

```
length : List a → Nat
length []      = 0
length (_::xs) = 1 + length xs
```

```
length : (xs : List a) → Nat
length []      = 0
length (_::xs) = 1 + length xs
```

Чуточку о синтаксисе

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

```
data Maybe : Type → Type where
  Nothing : Maybe a
  Just    : a → Maybe a
```

```
length : List a → Nat
length []      = 0
length (_::xs) = 1 + length xs
```

```
length : (xs : List a) → Nat
length []      = 0
length (_::xs) = 1 + length xs
```

```
elem : Eq a ⇒ a → List a → Bool
```

Чуточку о синтаксисе

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

```
data Maybe : Type → Type where
  Nothing : Maybe a
  Just    : a → Maybe a
```

```
length : List a → Nat
length []      = 0
length (_::xs) = 1 + length xs
```

```
length : (xs : List a) → Nat
length []      = 0
length (_::xs) = 1 + length xs
```

```
elem : Eq a ⇒ a → List a → Bool
elem : a → Eq a ⇒ List a → Bool
```

-- *

Чуточку о синтаксисе

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

```
data Maybe : Type → Type where
  Nothing : Maybe a
  Just    : a → Maybe a
```

```
length : List a → Nat
length []      = 0
length (_::xs) = 1 + length xs
```

```
length : (xs : List a) → Nat
length []      = 0
length (_::xs) = 1 + length xs
```

```
elem : Eq a ⇒ a → List a → Bool
elem : a → Eq a ⇒ List a → Bool                -- *
elem : a → List a → Eq a ⇒ Bool                -- **
```

Вы же знакомы с полиморфизмом?

```
id : a → a
id x = x
```

Вы же знакомы с полиморфизмом?

```
id : a → a
id x = x

id : forall a. a → a
id x = x
```

Первая ~~доза~~ зависимость

```
id : a → a
id x = x

id : forall a. a → a
id x = x

id : {0 a : Type} → a → a
id x = x
```

Первая ~~доза~~ зависимость

```
id : a → a
id x = x

id : forall a. a → a
id x = x

id : {0 a : Type} → a → a  -- всё ещё параметрична по `a`
id x = x
```

Первая ~~доза~~ зависимость

```
id : a → a
id x = x

id : forall a. a → a
id x = x

id : {0 a : Type} → a → a  -- всё ещё параметрична по `a`
id x = x


the : (0 a : Type) → a → a
the _ x = x
```

Первая ~~доза~~ зависимость

```
id : a → a
id x = x

id : forall a. a → a
id x = x

id : {0 a : Type} → a → a  -- всё ещё параметрична по `a`
id x = x


the : (0 a : Type) → a → a
the _ x = x
```

```
-- Haskell
someInt = 5 :: Int
```

Первая ~~доза~~ зависимость

```
id : a → a
id x = x

id : forall a. a → a
id x = x

id : {0 a : Type} → a → a  -- всё ещё параметрична по `a`
id x = x


the : (0 a : Type) → a → a
the _ x = x
```

```
-- Idris
someInt = the Int 5
```

```
-- Haskell
someInt = 5 :: Int
```

Дальше — больше

Дальше — больше

```
SomeStrangeType : Bool → Type
```

Дальше — больше

```
SomeStrangeType : Bool → Type
SomeStrangeType True  = String
SomeStrangeType False = Nat
```

Дальше — больше

```
SomeStrangeType : Bool → Type
SomeStrangeType True  = String
SomeStrangeType False = Nat


returns : (b : Bool) → SomeStrangeType b
```

Дальше — больше

```
SomeStrangeType : Bool → Type
SomeStrangeType True  = String
SomeStrangeType False = Nat


returns : (b : Bool) → SomeStrangeType b
returns True  = ?what
returns False = ?what_else
```

Дальше — больше

```
SomeStrangeType : Bool → Type
SomeStrangeType True  = String
SomeStrangeType False = Nat


returns : (b : Bool) → SomeStrangeType b
returns True  = "dependent types are cool"
returns False = ?what_else
```

Дальше — больше

```
SomeStrangeType : Bool → Type
SomeStrangeType True  = String
SomeStrangeType False = Nat


returns : (b : Bool) → SomeStrangeType b
returns True  = "dependent types are cool"
returns False = 42
```

Дальше — больше

```
SomeStrangeType : Bool -> Type
SomeStrangeType True  = String
SomeStrangeType False = Nat


returns : (b : Bool) -> SomeStrangeType b
returns True  = "dependent types are cool"
returns False = 42


takes : (b : Bool) -> SomeStrangeType b -> Integer
takes flag  val = ?what_type_of_val
```

Дальше — больше

```
SomeStrangeType : Bool -> Type
SomeStrangeType True  = String
SomeStrangeType False = Nat


returns : (b : Bool) -> SomeStrangeType b
returns True  = "dependent types are cool"
returns False = 42


takes : (b : Bool) -> SomeStrangeType b -> Integer
takes True  val = ?what_type_of_val_here
takes False val = ?what_type_of_val_there
```

Дальше — больше

```
SomeStrangeType : Bool -> Type
SomeStrangeType True  = String
SomeStrangeType False = Nat


returns : (b : Bool) -> SomeStrangeType b
returns True  = "dependent types are cool"
returns False = 42


takes : (b : Bool) -> SomeStrangeType b -> Integer
takes True  str = natToInteger $ length $ str ++ "cool?"
takes False val = ?what_type_of_val_there
```

Дальше — больше

```
SomeStrangeType : Bool -> Type
SomeStrangeType True  = String
SomeStrangeType False = Nat


returns : (b : Bool) -> SomeStrangeType b
returns True  = "dependent types are cool"
returns False = 42


takes : (b : Bool) -> SomeStrangeType b -> Integer
takes True  str = natToInteger $ length $ str ++ "cool?"
takes False 42  = ?something
takes False val = ?something_else
```

Дальше — больше

```
SomeStrangeType : Bool -> Type
SomeStrangeType True  = String
SomeStrangeType False = Nat


returns : (b : Bool) -> SomeStrangeType b
returns True  = "dependent types are cool"
returns False = 42


takes : (b : Bool) -> SomeStrangeType b -> Integer
takes True  str = natToInteger $ length $ str ++ "cool?"
takes False 42  = -64
takes False val = negate $ natToInteger val
```

Ещё дальше — ещё больше

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
head : Vect (1+n) a → a
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
total
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
total
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

```
data (·<·) : Nat → Nat → Type where
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
total
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

```
data (·<·) : Nat → Nat → Type where
  LTZ : 0 ·<· 1+n
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
total
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

```
data (·<·) : Nat → Nat → Type where
  LTZ : 0 ·<· 1+n
  LTS : n ·<· m → 1+n ·<· 1+m
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
total
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

```
data (·<·) : Nat → Nat → Type where
  LTZ : 0 ·<· 1+n
  LTS : n ·<· m → 1+n ·<· 1+m
```

```
index : (m : Nat) → Vect n a → m·<·n ⇒ a
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
total
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

```
data (·<·) : Nat → Nat → Type where
  LTZ : 0 ·<· 1+n
  LTS : n ·<· m → 1+n ·<· 1+m
```

```
index : (m : Nat) → Vect n a → m·<·n ⇒ a
index 0     (x::xs) = x
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
total
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

```
data (·<·) : Nat → Nat → Type where
  LTZ : 0 ·<· 1+n
  LTS : n ·<· m → 1+n ·<· 1+m
```

```
index : (m : Nat) → Vect n a → m·<·n ⇒ a
index 0     (x::xs) = x
index (S i) (x::xs) = index i xs
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
total
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

```
data (·<·) : Nat → Nat → Type where
  LTZ : 0 ·<· 1+n
  LTS : n ·<· m → 1+n ·<· 1+m
```

```
total
index : (m : Nat) → Vect n a → m·<·n ⇒ a
index 0     (x::xs) = x
index (S i) (x::xs) = index i xs
```

Ещё дальше — ещё больше

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  (::) : a → Vect n a → Vect (1+n) a
```

```
total
head : Vect (1+n) a → a
head (x::xs) = x
```

```
head' : Vect n a → Maybe a
head' []      = Nothing
head' (x::xs) = Just x
```

```
data Fin : Nat → Type where
  FZ : Fin (1+n)
  FS : Fin n → Fin (1+n)
```

```
-- натуральные числа,
-- строго меньшие `n`
```

```
total
index : Fin n → Vect n a → a
index 0      (x::xs) = x
index (FS i) (x::xs) = index i xs
```

Прыжок вдаль

Прыжок вдаль

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

          SortedBinTree a
```

Прыжок вдаль

```
data SortedBinTree : Type → Type where
  Empty : SortedBinTree a
  Node  : (x : a) → (left, right : SortedBinTree a) →
          -- all in left subtree < x, all in right subtree > x
          SortedBinTree a
```

Прыжок вдаль

```
data SortedBinTree : Type → Type where
  Empty : SortedBinTree a
  Node  : (x : a) → (left, right : SortedBinTree a) →
          -- all in left subtree < x, all in right subtree > x
          SortedBinTree a

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

Прыжок вдаль

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

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

Прыжок вдаль

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

data All : (a → Bool) → SortedBinTree a → Type where
  Empty' : All prop Empty
```

Прыжок вдаль

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

data All : (a → Bool) → SortedBinTree a → Type where
  Empty' : All prop Empty
  Node'  : forall prop.
           -- what?
           All prop $ Node x l r @{o} @{pl} @{pr}
```

Прыжок вдаль

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

data All : (a → Bool) → SortedBinTree a → Type where
  Empty' : All prop Empty
  Node'  : forall prop.
           -- `x` holds `prop`, ...
           All prop $ Node x l r @{o} @{pl} @{pr}
```

Прыжок вдаль

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

data All : (a → Bool) → SortedBinTree a → Type where
  Empty' : All prop Empty
  Node'  : forall prop.
           -- `x` holds `prop`, all in `l` and `r` holds `prop`
           All prop $ Node x l r @{o} @{pl} @{pr}
```

Прыжок вдаль

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

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

Улёт вдаль: целый язык

Улёт вдаль: целый язык

```
data Stmts : (funs  : List (Name, FunSig)) →
             (preV  : List (Name, Type)) →
             (postV : List (Name, Type)) → Type where
```

Улёт вдаль: целый язык

```
data Stmts : (funs  : List (Name, FunSig)) ->
             (preV  : List (Name, Type)) ->
             (postV : List (Name, Type)) -> Type where
```

```
(>>) : Stmts funs preV midV  -> Stmts funs midV postV ->
       Stmts funs preV postV
```

Улёт вдаль: целый язык

```
data Stmts : (funs  : List (Name, FunSig)) ->
             (preV  : List (Name, Type)) ->
             (postV : List (Name, Type)) -> Type where

  (.)  : (ty : Type) -> (n : Name) ->
         Stmts funs vars ((n, ty)::vars)

  (>>) : Stmts funs preV midV  -> Stmts funs midV postV ->
         Stmts funs preV postV
```

Улёт вдаль: целый язык

```
data Stmts : (funs  : List (Name, FunSig)) ->
             (preV  : List (Name, Type)) ->
             (postV : List (Name, Type)) -> Type where

  (.)  : (ty : Type) -> (n : Name) ->
         Stmts funs vars ((n, ty)::vars)

  (#=) : (n : Name) ->
         (v : Expr funs vars ?expr_ty) ->
         Stmts funs vars vars

  (>>) : Stmts funs preV midV  -> Stmts funs midV postV ->
         Stmts funs preV postV
```

Улёт вдаль: целый язык

```
data Stmts : (funs  : List (Name, FunSig)) ->
             (preV  : List (Name, Type)) ->
             (postV : List (Name, Type)) -> Type where

  (.)  : (ty : Type) -> (n : Name) ->
         Stmts funs vars ((n, ty)::vars)

  (#=) : (n : Name) -> (0 lk : n `IsIn` vars) =>
         (v : Expr funs vars lk.found) ->
         Stmts funs vars vars

  (>>) : Stmts funs preV midV  -> Stmts funs midV postV ->
         Stmts funs preV postV
```

Улёт вдаль: целый язык

```
data Stmts : (funs  : List (Name, FunSig)) ->
             (preV  : List (Name, Type)) ->
             (postV : List (Name, Type)) -> Type where

  (.)  : (ty : Type) -> (n : Name) ->
         Stmts funs vars ((n, ty)::vars)

  (#=) : (n : Name) -> (0 lk : n `IsIn` vars) =>
         (v : Expr funs vars lk.found) ->
         Stmts funs vars vars

  If   : (cond : Expr funs vars Bool) ->
         Stmts funs vars vThen -> Stmts funs vars vElse ->
         Stmts funs vars vars

  (>>) : Stmts funs preV midV  -> Stmts funs midV postV ->
         Stmts funs preV postV
```

Улёт вдаль: целый язык

```
data Expr : List (Name, FunSig) -> List (Name, Type) ->
            Type -> Type where
```

Улёт вдаль: целый язык

```
data Expr : List (Name, FunSig) -> List (Name, Type) ->
            Type -> Type where
  C : (x : ty) -> Expr funs vars ty
```

Улёт вдаль: целый язык

```
data Expr : List (Name, FunSig) -> List (Name, Type) ->
            Type -> Type where
  C : (x : ty) -> Expr funs vars ty
  V : (n : Name) -> (0 lk : n `IsIn` vars) =>
      Expr funs vars lk.found
```

Улёт вдаль: целый язык

```
record FunSig where
  constructor (⟹)
  From : List Type
  To   : Type

data Expr : List (Name, FunSig) → List (Name, Type) →
            Type → Type where

  C : (x : ty) → Expr funs vars ty

  V : (n : Name) → (0 lk : n `IsIn` vars) ⇒
      Expr funs vars lk.found
```

Улёт вдаль: целый язык

```
record FunSig where
  constructor (⟹)
  From : List Type
  To   : Type


data Expr : List (Name, FunSig) → List (Name, Type) →
            Type → Type where

  C : (x : ty) → Expr funs vars ty

  V : (n : Name) → (0 lk : n `IsIn` vars) ⇒
      Expr funs vars lk.found

  F : (n : Name) → (0 lk : n `IsIn` funs) ⇒
      All (Expr funs vars) lk.found.From →
      Expr funs vars lk.found.To
```

Улёт вдаль: целый язык

```
program : Stmts [] [] ?
program = do
  Int. "x"
  "x" #= C 5
```

Улёт вдаль: целый язык

```
program : Stmts [] [] ?
program = do
  Int. "x"
  "x" #= C 5
  Int. "y"; Bool. "res"
```

Улёт вдаль: целый язык

```
StdF : List (Name, FunSig)
StdF = [ ("+" , [Int, Int] ⟹ Int)
       , ("<" , [Int, Int] ⟹ Bool)
       , ("++", [Int] ⟹ Int)
       , ("||", [Bool, Bool] ⟹ Bool) ]
```

```
program : Stmts [] [] ?
program = do
  Int. "x"
  "x" #= C 5
  Int. "y"; Bool. "res"
```

Улёт вдаль: целый язык

```
StdF : List (Name, FunSig)
StdF = [ ("+" , [Int, Int] ⟹ Int)
       , ("<" , [Int, Int] ⟹ Bool)
       , ("++", [Int] ⟹ Int)
       , ("||", [Bool, Bool] ⟹ Bool) ]
```

```
program : Stmts StdF [] ?
program = do
  Int. "x"
  "x" #= C 5
  Int. "y"; Bool. "res"
```

Улёт вдаль: целый язык

```
StdF : List (Name, FunSig)
StdF = [ ("+" , [Int, Int] ⟹ Int)
       , ("<" , [Int, Int] ⟹ Bool)
       , ("++", [Int] ⟹ Int)
       , ("||", [Bool, Bool] ⟹ Bool) ]
```

```
program : Stmts StdF [] ?
program = do
  Int. "x"
  "x" #= C 5
  Int. "y"; Bool. "res"
  "y" #= F "+" [V "x", C 1]
```

Улёт вдаль: целый язык

```
StdF : List (Name, FunSig)
StdF = [ ("+" , [Int, Int] ⟹ Int)
       , ("<" , [Int, Int] ⟹ Bool)
       , ("++", [Int] ⟹ Int)
       , ("||", [Bool, Bool] ⟹ Bool) ]

program : Stmts StdF [] ?
program = do
  Int. "x"
  "x" #= C 5
  Int. "y"; Bool. "res"
  "y" #= F "+" [V "x", C 1]
  If (F "<" [F "++" [V "x"], V "y"])
     ?then_branch
     ?else_branch
```

Улёт вдаль: целый язык

```
StdF : List (Name, FunSig)
StdF = [ ("+" , [Int, Int] ⟹ Int)
       , ("<" , [Int, Int] ⟹ Bool)
       , ("++", [Int] ⟹ Int)
       , ("||", [Bool, Bool] ⟹ Bool) ]

program : Stmts StdF [] ?
program = do
  Int. "x"
  "x" #= C 5
  Int. "y"; Bool. "res"
  "y" #= F "+" [V "x", C 1]
  If (F "<" [F "++" [V "x"], V "y"])
     (do "y" #= C 0; "res" #= C False)
     ?else_branch
```

Улёт вдаль: целый язык

```
StdF : List (Name, FunSig)
StdF = [ ("+" , [Int, Int] ⟹ Int)
       , ("<" , [Int, Int] ⟹ Bool)
       , ("++", [Int] ⟹ Int)
       , ("||", [Bool, Bool] ⟹ Bool) ]

program : Stmts StdF [] ?
program = do
  Int. "x"
  "x" #= C 5
  Int. "y"; Bool. "res"
  "y" #= F "+" [V "x", C 1]
  If (F "<" [F "++" [V "x"], V "y"])
     (do "y" #= C 0; "res" #= C False)
     (do Int. "z"; "z" #= F "+" [V "x", V "y"]
         Bool. "b"; "b" #= F "<" [V "x", C 5]
         "res" #= F "||" [V "b", F "<" [V "z", C 6]])
```

Улёт вдаль: целый язык

```
failing "Mismatch between: Int and Bool"
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    Bool. "y"; "y" #= F "+" [V "x", C 1]
```

Улёт вдаль: целый язык

```
failing "Mismatch between: Int and Bool"
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    Bool. "y"; "y" #= F "+" [V "x", C 1]

failing "Mismatch between: [] and [Int]"
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    Int. "y"; "y" #= F "+" [V "x"]
```

Улёт вдаль: целый язык

```
failing "Mismatch between: Int and Bool"
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    Bool. "y"; "y" #= F "+" [V "x", C 1]

failing "Mismatch between: [] and [Int]"
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    Int. "y"; "y" #= F "+" [V "x"]

failing "Mismatch between: Bool and Int"
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    Int. "y"; "y" #= F "+" [C True, V "x"]
```

Улёт вдаль: целый язык

```
failing #"
    Can't find an implementation for IsIn "z" [("x", Int)]"#
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    "z" #= V "x"
```

Улёт вдаль: целый язык

```
failing #"
    Can't find an implementation for IsIn "z" [("x", Int)]"#
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"; "x" #= C 5
    "z" #= V "x"
```

```
failing #"
    Can't find an implementation for IsIn "z" [("x", Int)]"#
  bad : Stmts StdF [] ?
  bad = do
    Int. "x"
    "x" #= V "z"
```

Property-based testing

Property-based testing

- Тестирование функции/системы на *произвольном* входе

Property-based testing

- Тестирование функции/системы на *произвольном* входе
- Оценка, а не предугадывание результата

Property-based testing

- Тестирование функции/системы на *произвольном* входе
- Оценка, а не предугадывание результата
- Рандомизированная генерация входных значений

Property-based testing

- Тестирование функции/системы на *произвольном* входе
- Оценка, а не предугадывание результата
- Рандомизированная генерация входных значений
- (Иногда) автоматическая деривация генераторов

Property-based testing

- Тестирование функции/системы на *произвольном* входе
- Оценка, а не предугадывание результата
- Рандомизированная генерация входных значений
- (Иногда) автоматическая деривация генераторов
- (Иногда) автоматический shrinking

Property-based testing

- Тестирование функции/системы на *произвольном* входе
- Оценка, а не предугадывание результата
- Рандомизированная генерация входных значений
- (Иногда) автоматическая деривация генераторов
- (Иногда) автоматический shrinking
- QuickCheck, Hedgehog, Validity, ScalaCheck, ...

Property-based testing

```
insertOk : Property
insertOk = property $ do
  insert 2 [1, 3, 5] === [1, 2, 3, 5]
```

Property-based testing

```
insertOk : Property
insertOk = property $ do
  insert ?x ?xs === ?result
```

Property-based testing

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  insert x ?xs === ?result
```

Property-based testing

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  insert x xs === ?result
```

Property-based testing

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```

Property-based testing

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs
```

Property-based testing

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs


reverseAndConcat : Property
reverseAndConcat = property $ do
  xs <- forAll arbitraryList
  ys <- forAll arbitraryList
  reverse xs ++ reverse ys === reverse (ys ++ xs)
```

А что, если reverse = id?

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs


reverseAndConcat : Property
reverseAndConcat = property $ do
  xs <- forAll arbitraryList
  ys <- forAll arbitraryList
  reverse xs ++ reverse ys === reverse (ys ++ xs)
```

А что, если `reverse = id`?

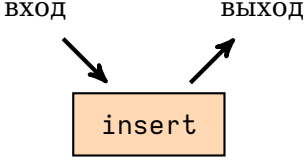
```
✗ reverseAndConcat failed after 13 tests.

  forAll 0 =
    [0]

  forAll 1 =
    [1]

  ─── Failed (- lhs) (+ rhs) ───
  - "[0, 1]"
  + "[1, 0]"
```

Property-based testing сверху

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```

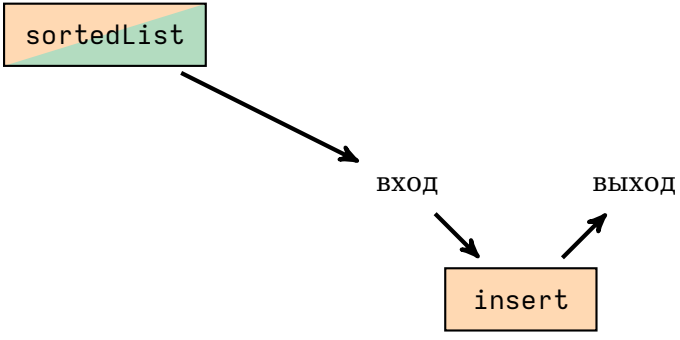
Property-based testing сверху

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```



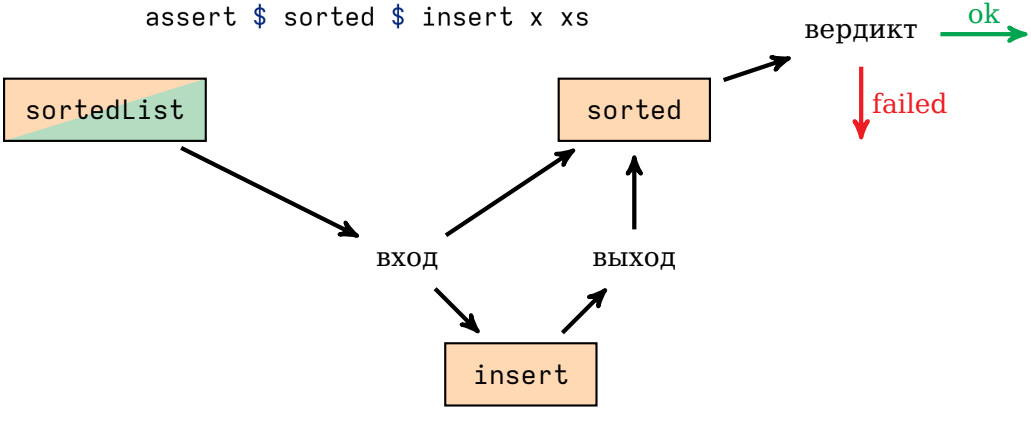
Property-based testing сверху

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```



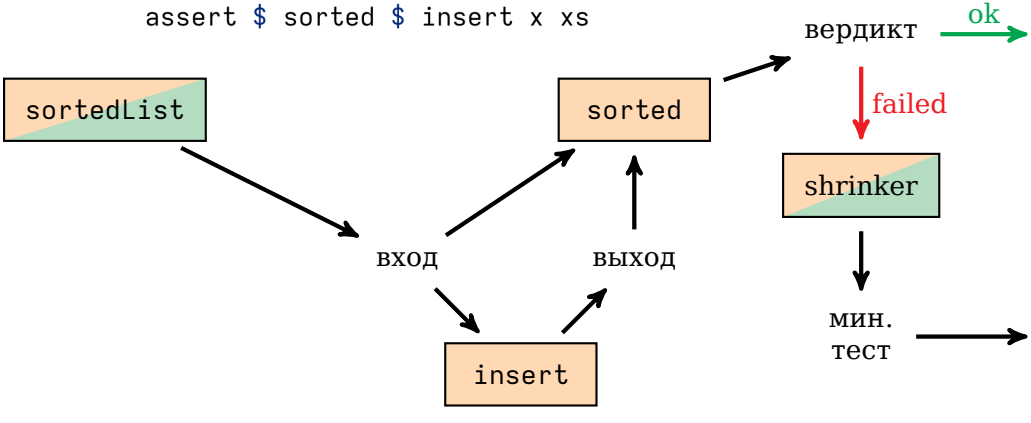
Property-based testing сверху

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```

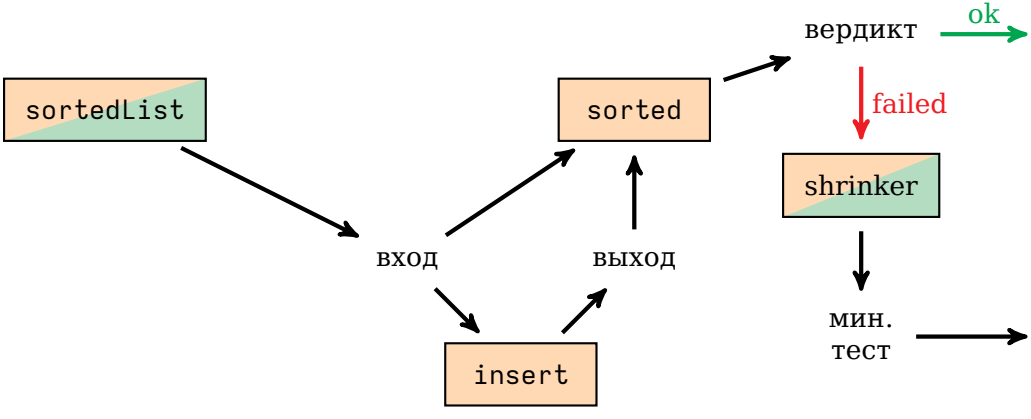


Property-based testing сверху

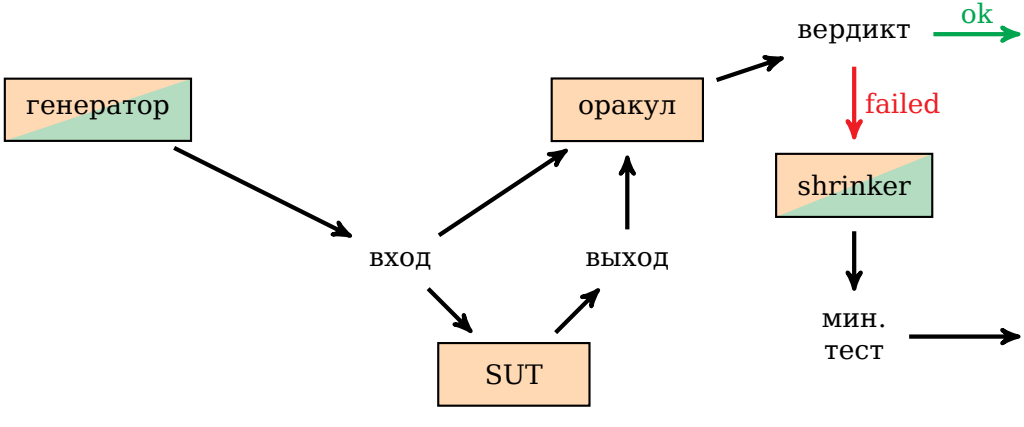
```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```



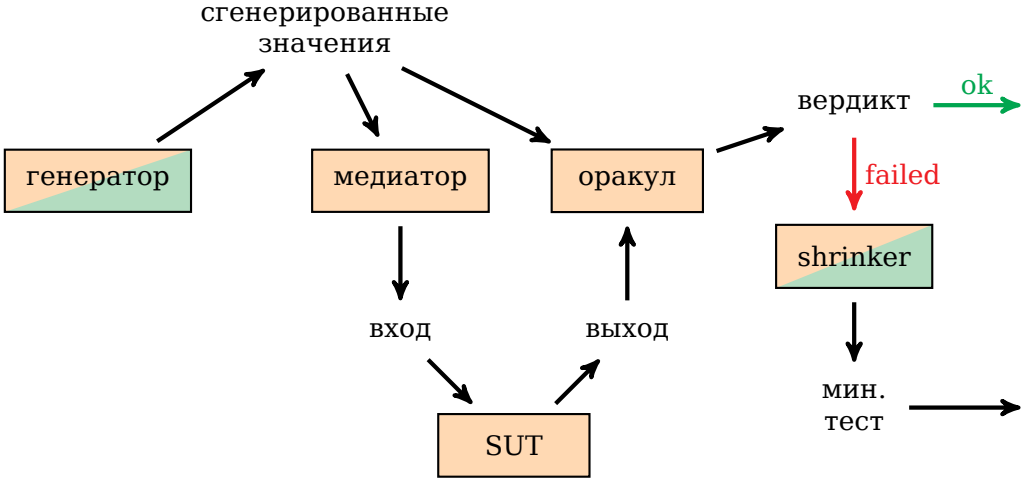
Property-based testing сверху

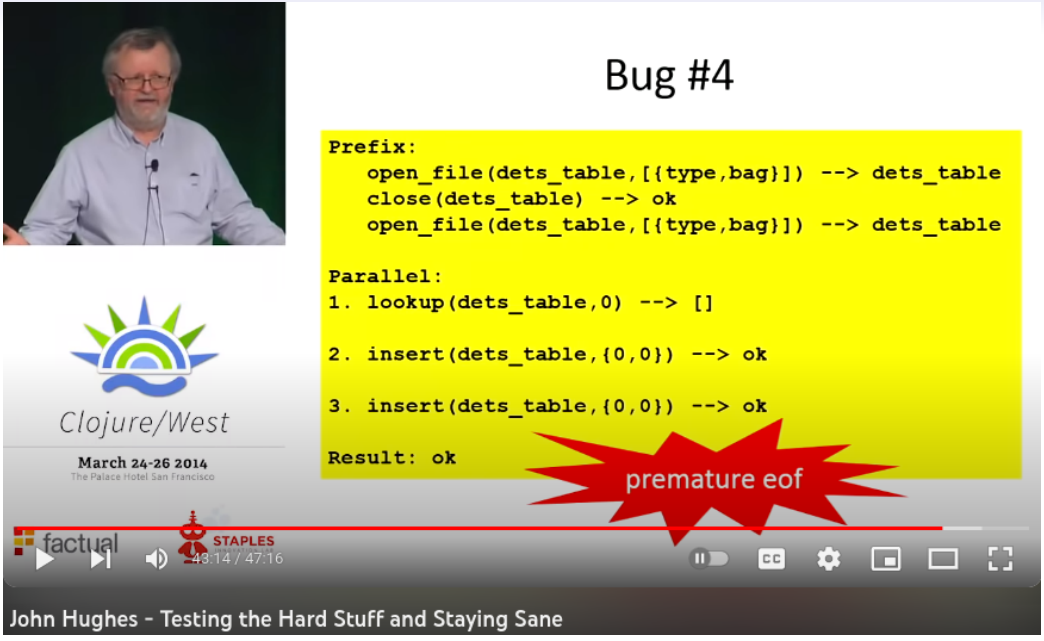


Property-based testing сверху



Property-based testing сверху





[1] https://www.youtube.com/watch?v=zi0rHwfiX1Q

Property-based testing

```
insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```

Property-based testing

```
sortedList : Gen $ List Nat


insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```

Property-based testing

```
sortedList : Gen $ List Nat
sortedList =
  foldr (\x, res ⇒ x :: map (+x) res) [] <$> arbitraryList

insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```

Зависимые типы + property-based testing

```
arbitrarySortedBinTree : Gen $ SortedBinTree Nat


sortedList : Gen $ List Nat
sortedList =
  foldr (\x, res ⇒ x :: map (+x) res) [] <$> arbitraryList

insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```

Зависимые типы + property-based testing

```
arbitrarySortedBinTree : Gen $ SortedBinTree Nat

sortedList : Gen $ List Nat
sortedList = toList <$> arbitrarySortedBinTree

insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```

Зависимые типы + property-based testing

```
arbitrarySortedBinTree : Gen $ SortedBinTree Nat
arbitrarySortedBinTree = %runElab deriveGen

sortedList : Gen $ List Nat
sortedList = toList <$> arbitrarySortedBinTree


insertOk : Property
insertOk = property $ do
  x  <- forAll arbitraryNat
  xs <- forAll sortedList
  assert $ sorted $ insert x xs
```

Зависимые типы + property-based testing

- Зависимые типы для описания сложных входных данных

Зависимые типы + property-based testing

- Зависимые типы для описания сложных входных данных
- Описать тип так, чтобы любое значение подходило

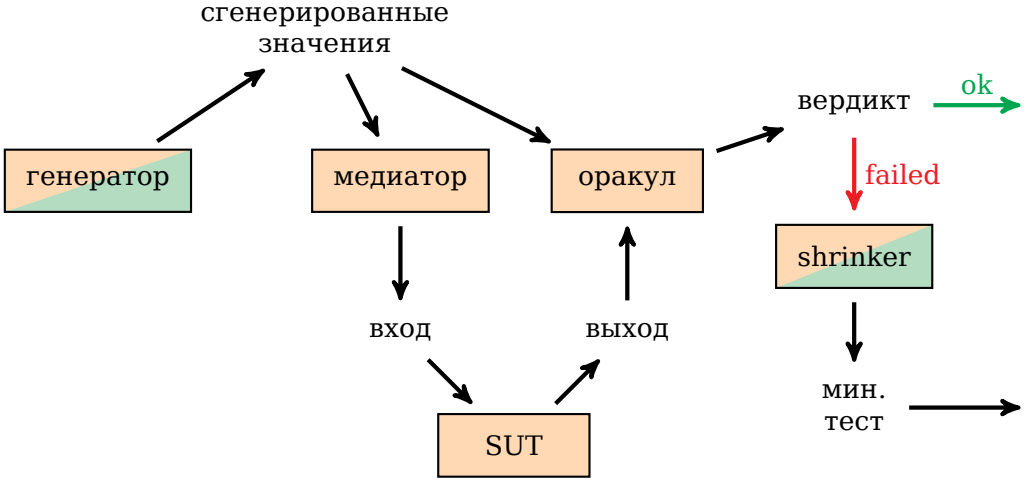
Зависимые типы + property-based testing

- Зависимые типы для описания сложных входных данных
- Описать тип так, чтобы любое значение подходило
- Возможность автоматической деривации генератора

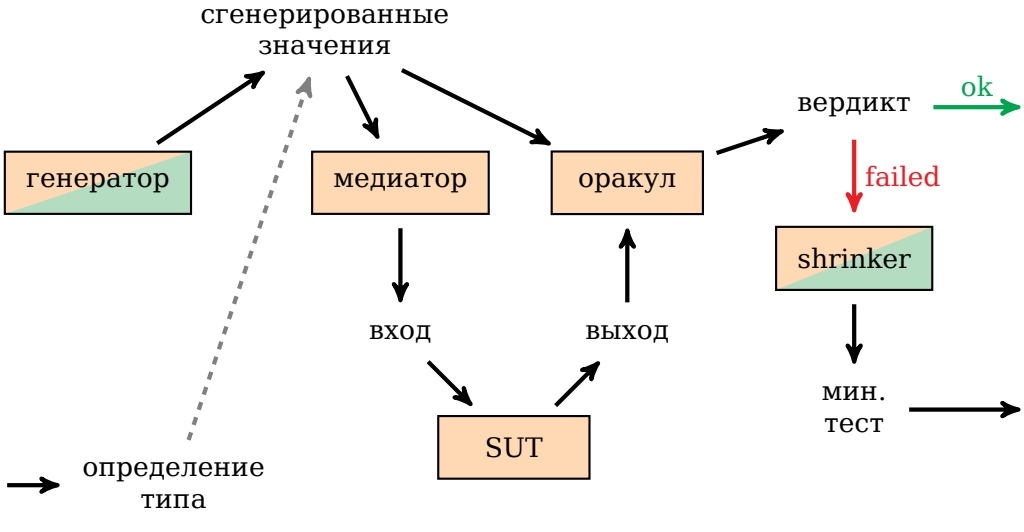
Зависимые типы + property-based testing

- Зависимые типы для описания сложных входных данных
- Описать тип так, чтобы любое значение подходило
- Возможность автоматической деривации генератора
- …
- Profit!

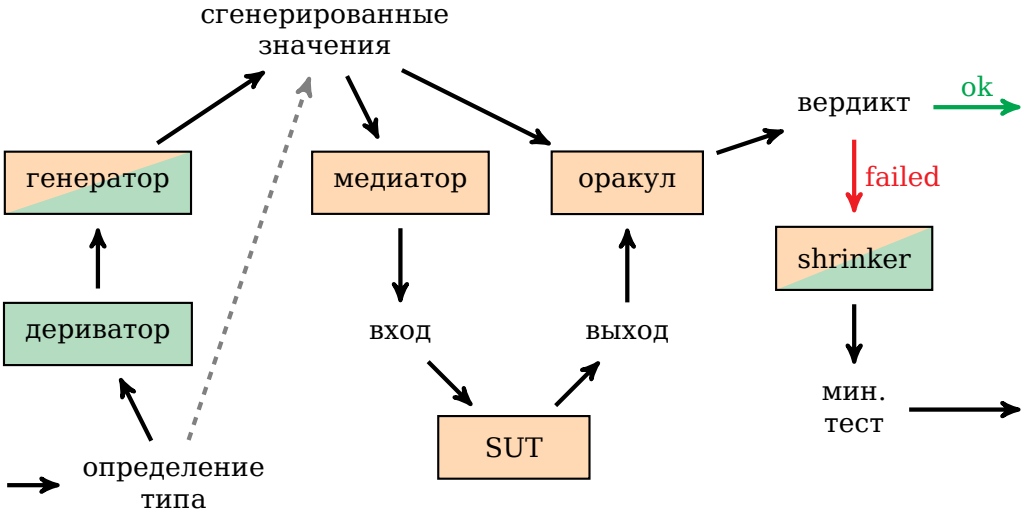
Property-based testing сверху



Property-based testing сверху



Property-based testing сверху



DepTyCheck (сейчас)

- Opensource-библиотека для PBT с зависимыми типами

DepTyCheck (сейчас)

- Opensource-библиотека для PBT с зависимыми типами
- Поддерживает деривацию генераторов

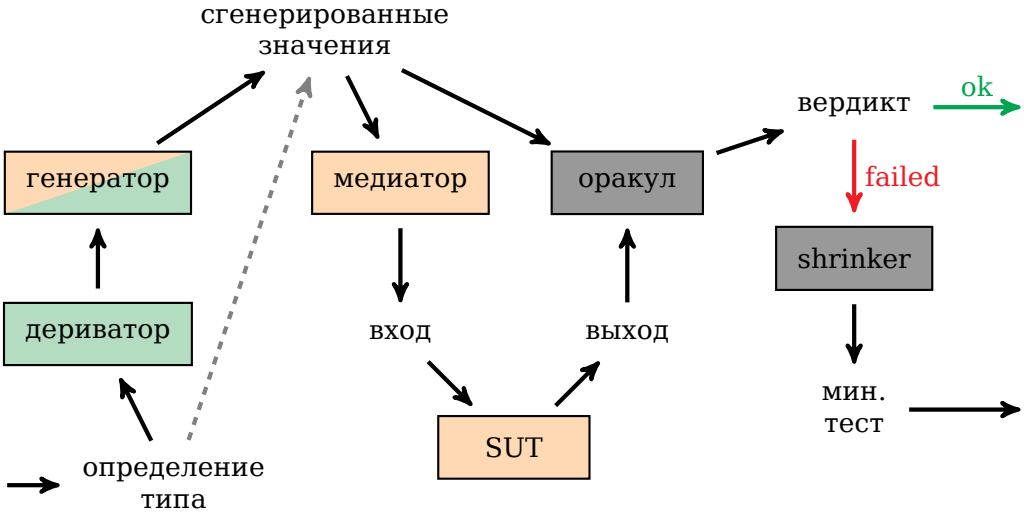
DepTyCheck (сейчас)

- Opensource-библиотека для PBT с зависимыми типами
- Поддерживает деривацию генераторов
- Гарантии полноты и распределения при деривации

DepTyCheck (сейчас)

- Opensource-библиотека для PBT с зависимыми типами
- Поддерживает деривацию генераторов
- Гарантии полноты и распределения при деривации
- Under heavy construction, очень многого ещё нет

DepTyCheck (сейчас)



Пример из жизни

Пример из жизни

- Диалект Typescript

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор
- Свежая промышленная разработка

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор
- Свежая промышленная разработка
- Специфировали подмножество

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор
- Свежая промышленная разработка
- Специфировали подмножество
  - Завершающиеся программы

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор
- Свежая промышленная разработка
- Специфировали подмножество
  - Завершающиеся программы
  - Циклы, ветвления, присваивания, исключения

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор
- Свежая промышленная разработка
- Специфировали подмножество
  - Завершающиеся программы
  - Циклы, ветвления, присваивания, исключения
  - Классы без методов, числа, массивы

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор
- Свежая промышленная разработка
- Специфировали подмножество
  - Завершающиеся программы
  - Циклы, ветвления, присваивания, исключения
  - Классы без методов, числа, массивы
- Наша спецификация

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор
- Свежая промышленная разработка
- Специфировали подмножество
  - Завершающиеся программы
  - Циклы, ветвления, присваивания, исключения
  - Классы без методов, числа, массивы
- Наша спецификация
  - Описание семантически корректных программ из подмножества

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор
- Свежая промышленная разработка
- Специфировали подмножество
  - Завершающиеся программы
  - Циклы, ветвления, присваивания, исключения
  - Классы без методов, числа, массивы
- Наша спецификация
  - Описание семантически корректных программ из подмножества
  - ~330 строк кода спецификация языка + столько же обвязка

Пример из жизни

- Диалект Typescript
- Имеет интерпретатор с JIT и компилятор
- Свежая промышленная разработка
- Специфировали подмножество
  - Завершающиеся программы
  - Циклы, ветвления, присваивания, исключения
  - Классы без методов, числа, массивы
- Наша спецификация
  - Описание семантически корректных программ из подмножества
  - ~330 строк кода спецификация языка + столько же обвязка
  - Частично деривированные, частично рукописные генераторы

Пример из жизни

Testing...

Пример из жизни

```
ASSERTION FAILED: false
IN /.../optimizations/lse.cpp:851: GetEliminationCode
Backtrace [tid=2990599]:
#0 : 0x7f876a1776c0 PrintStack(std::ostream&)
#1 : 0x7f876a177562 debug::AssertionFail(...)
#2 : 0x7f8767185e10 compiler::Lse::GetEliminationCode(...)
#3 : 0x7f8767186a20 compiler::Lse::DeleteInstructions(...)
...
```

Пример из жизни

Пример из жизни

Shrinking...

Пример из жизни

```
class C0 {
  x0: boolean
}

function main() : void {
  let x1: C0 = {x0: true}
  while(x1.x0) {
    x1.x0 = x1.x0
    x1.x0 = false
  }
}
```

Пример из жизни

Testing...

Пример из жизни

```
Wrong input 0 type 'i32' for inst:
    52.ref  NullCheck  v42, v51 -> (v55, v53)
                                        bc: 0x0000005d
ASSERTION FAILED: CheckType(GetInputType(inst, 0), ...)
IN /.../inst_checker_gen.h:694: VisitNullCheck
ERRNO: 29 (Illegal seek)
Backtrace [tid=3853514]:
#0 : 0x7f46fc7b393c PrintStack(std::ostream&)
#1 : 0x7f46fc7b37de debug::AssertionFail(...)
#2 : 0x7f46fe3760ad compiler::InstChecker::VisitNullCheck(...)
#3 : 0x7f46fe38dae5 compiler::InstChecker::VisitGraph()
#4 : 0x7f46fe35e63e compiler::InstChecker::Run(...)
#5 : 0x7f46fe33c1b2 compiler::GraphChecker::Check()
...
```

Пример из жизни

```
Wrong input 0 type 'i32' for inst:
    52.ref  NullCheck  v42, v51 -> (v55, v53)
                                        bc: 0x0000005d
ASSERTION FAILED: CheckType(GetInputType(inst, 0), ...)
IN /.../inst_checker_gen.h:694: VisitNullCheck
ERRNO: 29 (Illegal seek)
Backtrace [tid=3853514]:
#0 : 0x7f46fc7b393c PrintStack(std::ostream&)
#1 : 0x7f46fc7b37de debug::AssertionFail(...)
#2 : 0x7f46fe3760ad compiler::InstChecker::VisitNullCheck(...)
#3 : 0x7f46fe38dae5 compiler::InstChecker::VisitGraph()
#4 : 0x7f46fe35e63e compiler::InstChecker::Run(...)
#5 : 0x7f46fe33c1b2 compiler::GraphChecker::Check()
...
```

Shrinking...

Пример из жизни

```
function main() {
   for(let x2 of [0]) {
     let x3: boolean = false
     for(let x4 of [0]) {
       let x5: int[][] = [[]]
       let fuel1 = 0
     }
   }
   let fuel0 = 0
}
```

Пример из жизни

Testing...

Пример из жизни

```
ASSERTION FAILED: block->GetGraph() == GetGraph()
IN /.../optimizer/ir/graph_cloner.h:176: GetClone
Backtrace [tid=2902033]:
#0 : 0x7fe71892b820 PrintStack(std::ostream&)
#1 : 0x7fe71892b6c2 debug::AssertionFail(...)
#2 : 0x7fe71a61ae61 compiler::GraphCloner::GetClone(...)
#3 : 0x7fe71a61162a compiler::GraphCloner::CopyLoop(...)
#4 : 0x7fe71a611839 compiler::GraphCloner::CopyLoop(...)
#5 : 0x7fe71a611173 compiler::GraphCloner::CloneAnalyses(...)
#6 : 0x7fe71a610d1f compiler::GraphCloner::CloneGraph()
#7 : 0x7fe71a5b377c compiler::GraphChecker::GraphChecker(...)
...
```

Пример из жизни

```
ASSERTION FAILED: block->GetGraph() == GetGraph()
IN /.../optimizer/ir/graph_cloner.h:176: GetClone
Backtrace [tid=2902033]:
#0 : 0x7fe71892b820 PrintStack(std::ostream&)
#1 : 0x7fe71892b6c2 debug::AssertionFail(...)
#2 : 0x7fe71a61ae61 compiler::GraphCloner::GetClone(...)
#3 : 0x7fe71a61162a compiler::GraphCloner::CopyLoop(...)
#4 : 0x7fe71a611839 compiler::GraphCloner::CopyLoop(...)
#5 : 0x7fe71a611173 compiler::GraphCloner::CloneAnalyses(...)
#6 : 0x7fe71a610d1f compiler::GraphCloner::CloneGraph()
#7 : 0x7fe71a5b377c compiler::GraphChecker::GraphChecker(...)
...
```

Shrinking...

Пример из жизни

```
class C0 {
  x0: boolean

  f() : string {
    return ""
  }
}
```

```
function main() : void {
  let x2: C0 = {x0: true}
  let fuel0 = 1
  while(fuel0 > 0) {
    do {
      fuel0--
      do {
        fuel0--
        let s = x2.f()
      } while(true && (fuel0 > 0))
    } while(true && (fuel0 > 0))
  }
}
```

Пример из жизни

Testing...

Пример из жизни

```
TypeError: Unreachable statement. [<filename>:26:34]
```

Пример из жизни

```
TypeError: Unreachable statement. [<filename>:26:34]
```

Shrinking...

Пример из жизни

```
TypeError: Unreachable statement. [<filename>:3:30]
```

```
function main() : void {
  let x1: Int = 1
  while(([false, true])[x1]) {
  }
}
```

Пример из жизни

- ...и так далее

Пример из жизни

- …и так далее
- Было найдено 9 подобных ошибок
  - В JIT-, AOT-оптимизаторе, тайпчекере, кодогенераторе

Пример из жизни

- …и так далее
- Было найдено 9 подобных ошибок
  - В JIT-, AOT-оптимизаторе, тайпчекере, кодогенераторе
- Ещё 8 во время написания спецификации

Выводы

Выводы

- Зависимые типы можно применять для спецификации сложных входных данных!

Выводы

- Зависимые типы можно применять для спецификации сложных входных данных!
- Ошибки у вполне тестированных систем лежат в пересечении множества фич

Выводы

- Зависимые типы можно применять для спецификации сложных входных данных!
- Ошибки у вполне тестированных систем лежат в пересечении множества фич
- Одна спецификация находит много ошибок

Выводы

- Зависимые типы можно применять для спецификации сложных входных данных!
- Ошибки у вполне тестированных систем лежат в пересечении множества фич
- Одна спецификация находит много ошибок
- Нацеленность не всегда полезна, ошибки могут быть не там, где мы их ожидаем увидеть

Выводы

- Зависимые типы можно применять для спецификации сложных входных данных!
- Ошибки у вполне тестированных систем лежат в пересечении множества фич
- Одна спецификация находит много ошибок
- Нацеленность не всегда полезна, ошибки могут быть не там, где мы их ожидаем увидеть
- Property-based testing классный

Выводы

- Зависимые типы можно применять для спецификации сложных входных данных!
- Ошибки у вполне тестированных систем лежат в пересечении множества фич
- Одна спецификация находит много ошибок
- Нацеленность не всегда полезна, ошибки могут быть не там, где мы их ожидаем увидеть
- Property-based testing классный
- Зависимые типы классные ;-)

Выводы

- Зависимые типы можно применять для спецификации сложных входных данных!
- Ошибки у вполне тестированных систем лежат в пересечении множества фич
- Одна спецификация находит много ошибок
- Нацеленность не всегда полезна, ошибки могут быть не там, где мы их ожидаем увидеть
- Property-based testing классный
- Зависимые типы классные ;-)
- А вместе – так совсем улёт

Если стало интересно

Эта презентация

DepTyCheck

Код со слайдов

Спасибо!

Эта презентация

DepTyCheck

Код со слайдов

Вопросы?