

Property-based testing и зависимые типы

и немножко функционального программирования

Денис Буздáлов

11 февраля 2026



О докладе

Сперва
●○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

О докладе

- Познавательно-развлекательный

О докладе

- Познавательно-развлекательный
- Хотелось познакомить с конкретной работой

О докладе

- Познавательно-развлекательный
- Хотелось познакомить с конкретной работой
- Придётся знакомиться с целой областью

О докладе

- Познавательно-развлекательный
- Хотелось познакомить с конкретной работой
- Придётся знакомиться с целой областью
- Упрощения, заметания под ковёр

О докладе

- Познавательно-развлекательный
- Хотелось познакомить с конкретной работой
- Придётся знакомиться с целой областью
- Упрощения, заметания под ковёр
- Если покажется, что всё понятно в чём-то новом, это только кажется ;-)

О докладе

- Познавательно-развлекательный
- Хотелось познакомить с конкретной работой
- Придётся знакомиться с целой областью
- Упрощения, заметания под ковёр
- Если покажется, что всё понятно в чём-то новом, это только кажется ;-)
- Будет много кода разных синтаксисов

О докладе

- Познавательно-развлекательный
- Хотелось познакомить с конкретной работой
- Придётся знакомиться с целой областью
- Упрощения, заметания под ковёр
- Если покажется, что всё понятно в чём-то новом, это только кажется ;-)
- Будет много кода разных синтаксисов
- Вопросы по существу стоит задавать сразу

Сперва
●

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Что такое *хорошее тестирование*?

Сперва
●

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Что такое *хорошее тестирование*?

- Формальная верификация? ;-)

Что такое *хорошее тестирование*?

- Формальная верификация? ;-)
- Находит баги?

Что такое *хорошее тестирование*?

- Формальная верификация? ;-)
- Находит баги?
- Покрывает код?

Что такое *хорошее тестирование*?

- Формальная верификация? ;-)
- Находит баги?
- Покрывает код?
- Покрывает функциональность/требования?

Что такое *хорошее тестирование*?

- Формальная верификация? ;-)
- Находит баги?
- Покрывает код?
- Покрывает функциональность/требования?
- Повышает уверенность?

Что такое *хорошее* тестирование?

- Формальная верификация? ;-)
- Находит баги?
- Покрывает код?
- Покрывает функциональность/требования?
- Повышает уверенность?
- Много тестов?

Что такое *хорошее тестирование*?

- Формальная верификация? ;-)
- Находит баги?
- Покрывает код?
- Покрывает функциональность/требования?
- Повышает уверенность?
- Много тестов?
- Быстро работает?

Что такое *хорошее* тестирование?

- Формальная верификация? ;-)
- Находит баги?
- Покрывает код?
- Покрывает функциональность/требования?
- Повышает уверенность?
- Много тестов?
- Быстро работает?
- Легко разрабатывается?

Что такое *хорошее* тестирование?

- Формальная верификация? ;-)
- Находит баги?
- Покрывает код?
- Покрывает функциональность/требования?
- Повышает уверенность?
- Много тестов?
- Быстро работает?
- Легко разрабатывается?
- Хотя бы присутствует?

Что такое *хорошее* тестирование?

- Формальная верификация? ;-)
- Находит баги?
- Покрывает код?
- Покрывает функциональность/требования?
- Повышает уверенность?
- Много тестов?
- Быстро работает?
- Легко разрабатывается?
- Хотя бы присутствует?
- Создаёт ситуации, о которых автор тестов даже не думал?

Property-based testing

Property-based testing

- Тестирование функции/системы на *произвольном* интересном входе

Property-based testing

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

Property-based testing

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

Property-based testing

- Тестирование функции/системы на *произвольном* интересном входе
- Оценка, а не предугадывание результата
- Рандомизированная генерация входных значений
- Десятки библиотек под множество языков

Haskell, Erlang, Scala, Python, Coq, Idris, C#, C++, Clojure, D, Elixir, Elm, F#, Go, Java, JavaScript, Julia, Kotlin, Nim, OCaml, Prolog, Racket, Ruby, Rust, Swift, TypeScript, ...

Свойство insert?

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

Свойство insert?

`insertOk` : `Property`

`insertOk` = `property` \$ `do`

`insert` 2 [1, 3, 5] == [1, 2, 3, 5]

`def insertOk():`

`xs = [1, 3, 5]`

`insert(xs, 2)`

`assert xs == [1, 2, 3, 5]`

Свойство insert?

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

```
def insertOk(x, xs):
  insert(xs, x)
```

Свойство insert?

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

```
@given(st.integers())
def insertOk(x, xs):
  insert(xs, x)
```

Свойство insert?

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  insert x xs == ?result
```

```
@given(st.integers(),
        sortedLists(st.integers()))
def insertOk(x, xs):
  insert(xs, x)
```

Свойство insert!

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  assert $ sorted $ insert x xs
```

```
@given(st.integers(),
  sortedLists(st.integers()))
def insertOk(x, xs):
  insert(xs, x)
  assert is_sorted(xs)
```

Свойство insert!

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs
```

```
@given(st.integers(),
       sortedLists(st.integers()))
def insertOk(x, xs):
  insert(xs, x)
  assert is_sorted(xs)
  assert x in xs
```


Свойство insert!

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs
```

Свойство insert!

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs
```

```
— sorted list insertion —
✓ insertOk passed 100 tests.
```

Свойство insert!

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs
```

А что, если `insert x xs = x :: xs`?

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs
```

А что, если `insert x xs = x :: xs`?

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs
```

```
— sorted list insertion —
✗ insertOk failed after 14 tests.
```

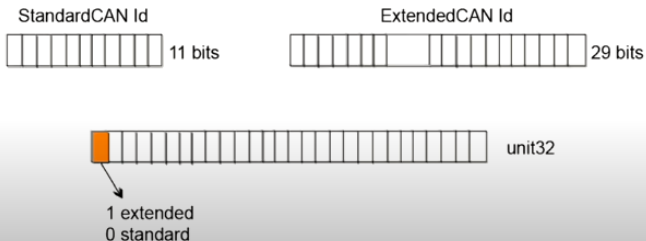
```
forAll 0 =
  1
```

```
forAll 1 =
  [0]
```



The Problem

CAN bus identifiers determine bus priority



Clojure/West

March 24-26 2014

The Palace Hotel San Francisco



John Hughes - Testing the Hard Stuff and Staying Sane

¹<https://www.youtube.com/watch?v=zi0rHwfiX1Q>



Clojure/West

March 24-26 2014
The Palace Hotel San Francisco



STAPLES

48:14 / 47:16

Bug #4

Prefix:

```
open_file(dets_table, [{type, bag}]) --> dets_table  
close(dets_table) --> ok  
open_file(dets_table, [{type, bag}]) --> dets_table
```

Parallel:

1. lookup(dets_table, 0) --> []
2. insert(dets_table, {0, 0}) --> ok
3. insert(dets_table, {0, 0}) --> ok

Result: ok

premature eof

John Hughes - Testing the Hard Stuff and Staying Sane

¹<https://www.youtube.com/watch?v=zi0rHwfiX1Q>

Сперва
○○

Property-based
○○●○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Хорошо применённый property-based testing

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы
- ✓ может найти практически ненаходимое ручным тестированием

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы
- ✓ может найти практически ненаходимое ручным тестированием
- ✓ находит то, о чём не подозревали, *хороший* метод

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы
- ✓ может найти практически ненаходимое ручным тестированием
- ✓ находит то, о чём не подозревали, *хороший* метод
- ✓ сложность тестирования растёт медленнее сложности SUT

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы
- ✓ может найти практически ненаходимое ручным тестированием
- ✓ находит то, о чём не подозревали, *хороший* метод
- ✓ сложность тестирования растёт медленнее сложности SUT
- ✗ надо уметь выбирать подходящие свойства

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы
- ✓ может найти практически ненаходимое ручным тестированием
- ✓ находит то, о чём не подозревали, *хороший* метод
- ✓ сложность тестирования растёт медленнее сложности SUT
- ✗ надо уметь выбирать подходящие свойства
- ✗ reimplementation trap

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы
- ✓ может найти практически ненаходимое ручным тестированием
- ✓ находит то, о чём не подозревали, *хороший* метод
- ✓ сложность тестирования растёт медленнее сложности SUT
- ✗ надо уметь выбирать подходящие свойства
- ✗ reimplementation trap
- ✗ формализация требований, нужен опыт и mindsetting

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы
- ✓ может найти практически ненаходимое ручным тестированием
- ✓ находит то, о чём не подозревали, *хороший* метод
- ✓ сложность тестирования растёт медленнее сложности SUT
- ✗ надо уметь выбирать подходящие свойства
- ✗ reimplementation trap
- ✗ формализация требований, нужен опыт и mindsetting
 - ✓ инварианты, модели, метаморфное тестирование, автоматы

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы
- ✓ может найти практически ненаходимое ручным тестированием
- ✓ находит то, о чём не подозревали, *хороший* метод
- ✓ сложность тестирования растёт медленнее сложности SUT
- ✗ надо уметь выбирать подходящие свойства
- ✗ reimplementation trap
- ✗ формализация требований, нужен опыт и mindsetting
 - ✓ инварианты, модели, метаморфное тестирование, автоматы
- ✗ написание генераторов, корректность, полнота, распределения

Хорошо применённый property-based testing

- ✓ одна спецификация находит ошибки в разных местах
- ✓ высокоуровневая спецификация может находить низкоуровневые проблемы
- ✓ может найти практически ненаходимое ручным тестированием
- ✓ находит то, о чём не подозревали, *хороший* метод
- ✓ сложность тестирования растёт медленнее сложности SUT
- ✗ надо уметь выбирать подходящие свойства
- ✗ reimplementation trap
- ✗ формализация требований, нужен опыт и mindsetting
 - ✓ инварианты, модели, метаморфное тестирование, автоматы
- ✗ написание генераторов, корректность, полнота, распределения
 - ✓ деривация

Деривация?

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs
```

Деривация?

```
arbitraryNat : Gen Nat
```

```
sortedNatList : Gen (List Nat)
```

```
insertOk : Property
insertOk = property $ do
  x ← forAll arbitraryNat
  xs ← forAll sortedNatList
  assert $ sorted $ insert x xs
  assert $ x `elem` insert x xs
```

Деривация?

```
arbitraryNat : Gen Nat  
arbitraryNat = deriveGen
```

```
sortedNatList : Gen (List Nat)
```

```
insertOk : Property  
insertOk = property $ do  
  x ← forAll arbitraryNat  
  xs ← forAll sortedNatList  
  assert $ sorted $ insert x xs  
  assert $ x `elem` insert x xs
```

Деривация?

```
arbitraryNat : Gen Nat  
arbitraryNat = deriveGen
```

```
arbitraryNatList : Gen (List Nat)
```

```
sortedNatList : Gen (List Nat)
```

```
insertOk : Property  
insertOk = property $ do  
  x ← forAll arbitraryNat  
  xs ← forAll sortedNatList  
  assert $ sorted $ insert x xs  
  assert $ x `elem` insert x xs
```

Деривация?

```
arbitraryNat : Gen Nat
arbitraryNat = deriveGen
```

```
arbitraryNatList : Gen (List Nat)
```

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

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


Деривация?

```
arbitraryNat : Gen Nat
arbitraryNat = deriveGen
```

```
arbitraryNatList : Gen (List Nat)
arbitraryNatList = deriveGen
```

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

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

Намерения выражены по-разному

```
arbitraryNat : Gen Nat
```

```
arbitraryNat = deriveGen
```

— намерения выражены типом

```
arbitraryNatList : Gen (List Nat)
```

```
arbitraryNatList = deriveGen
```

— намерения выражены типом

```
sortedNatList : Gen (List Nat)
```

```
sortedNatList =  
  foldr (\x, res => x :: map (+x) res) [] <$> arbitraryNatList
```

— намерения выражены исполняемым кодом

```
insertOk : Property
```

```
insertOk = property $ do
```

```
  x <- forAll arbitraryNat
```

```
  xs <- forAll sortedNatList
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```

Деривация понимает типы

```
arbitraryNat : Gen Nat
```

```
arbitraryNat = deriveGen
```

— намерения выражены типом

```
arbitraryNatList : Gen (List Nat)
```

```
arbitraryNatList = deriveGen
```

— намерения выражены типом

```
sortedNatList : Gen (List Nat)
```

```
sortedNatList =  
  foldr (\x, res => x :: map (+x) res) [] <$> arbitraryNatList
```

— намерения выражены исполняемым кодом

```
insertOk : Property
```

```
insertOk = property $ do
```

```
  x <- forAll arbitraryNat
```

```
  xs <- forAll sortedNatList
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```

Сперва
○○

Property-based
○○○○○

Type-driven
●○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Type-driven development

Type-driven development

- тратим усилия, чтобы выразить *намерения* через типы

Type-driven development

- тратим усилия, чтобы выразить *намерения* через типы
- получаем при должном старании
 - меньше некорректных реализаций
 - проще логика обработки
 - меньше ненужных проверок или кода обработки
 - помощь от компилятора и тулинга
 - ...

Type-driven development

- тратим усилия, чтобы выразить *намерения* через типы
- получаем при должном старании
 - меньше некорректных реализаций
 - проще логика обработки
 - меньше ненужных проверок или кода обработки
 - помощь от компилятора и тулинга
 - ...
- возможности зависят от выразительности системы типов

Type-driven property-based testing

- тратим усилия, чтобы выразить *намерения* через типы
- получаем при должном старании
 - меньше некорректных реализаций
 - проще логика обработки
 - меньше ненужных проверок или кода обработки
 - помощь от компилятора и тулинга
 - ...
 - мощные хорошие тесты
- возможности зависят от выразительности системы типов

Например, меньше некорректных реализаций

```
typedef time_t Time;
```

Например, меньше некорректных реализаций

```
typedef time_t Time;
```

```
void f(Time t, Time f, int a) {  
    Time x = t + f;  
    Time y = t * a;  
    // ...используем `x` и `y` ...  
}
```

Например, меньше некорректных реализаций

```
typedef time_t Time;
```

```
void f(Time t, Time f, int a) {  
    Time x = t + f;  
    Time y = t * f;  
    // ...используем `x` и `y` ...  
}
```

Например, меньше некорректных реализаций

```
struct Time {  
    time_t sec;  
    Time(time_t s) { sec = s; }  
    Time operator+(const Time &t) {  
        return Time(sec + t.sec);  
    }  
    Time operator*(const int n) {  
        return Time(sec * n);  
    }  
};
```

```
void f(Time t, Time f, int a) {  
    Time x = t + f;  
    Time y = t * f;  
    // ...используем `x` и `y` ...  
}
```

Например, меньше некорректных реализаций

```
struct Time {
```

```
x.cpp: In function 'void f(Time, Time, int)':
```

```
x.cpp:16:14: error: no match for 'operator*'
```

```
(operand types are 'Time' and 'Time')
```

```
16 |     Time y = t * f;
    |               ~ ^ ~
    |               | |
    |               | Time
    |               Time
```

```
void f(Time t, Time f, int a) {
```

```
    Time x = t + f;
```

```
    Time y = t * f; // Boom!
```

```
    // ...используем `x` и `y` ...
```

```
}
```

Например, меньше некорректных реализаций

```
struct Time {
```

```
x.cpp: In function 'void f(Time, Time, int)':
```

```
x.cpp:16:14: error: no match for 'operator*'
```

```
(operand types are 'Time' and 'Time')
```

```
16 |     Time y = t * f;  
    |               ~ ^ ~  
    |               |  |  
    |               |  Time  
    |               Time
```

```
void f(Time t, Time f, int a) {  
    Time x = t + f;  
    Time y = t * f; // Boom!  
    // ...используем `x` и `y` ...  
}
```

Не подумайте, что я считаю систему типов C++ *хорошей*. Но пример показательный

Например, проще логика обработки

```
def maximum(vs: List[Int]): Int =  
  vs match  
    case Nil => ???  
    case head::rest => ???
```

Например, проще логика обработки

```
def maximum(vs: List[Int]): Int =  
  vs match  
    case Nil => ???  
    case head::rest => rest.fold(head)(_ max _)
```


Например, проще логика обработки

```
def maximum(vs: List[Int]): Int =  
  vs match  
    case Nil => -1  
    case head::rest => rest.fold(head)(_ max _)
```

Например, проще логика обработки

```
def maximum(vs: List[Int]): Int =  
  vs match  
    case Nil => -1  
    case head::rest => rest.fold(head)(_ max _)
```

// но результат `-1` может быть и для непустого списка

Например, проще логика обработки

```
def maximum(vs: List[Int]): Int =  
  vs match  
    case Nil ⇒ Int.MaxValue  
    case head::rest ⇒ rest.fold(head)(_ max _)
```

Например, проще логика обработки

```
def maximum(vs: List[Int]): Int =  
  vs match  
    case Nil ⇒ Int.MaxValue  
    case head::rest ⇒ rest.fold(head)(_ max _)
```

// всё равно не можем отличить две ситуации по результату

Например, проще логика обработки

```
def maximum(vs: List[Int]): Int =  
  vs match  
    case Nil ⇒ throw new IndexOutOfBoundsException()  
    case head::rest ⇒ rest.fold(head)(_ max _)
```

Например, проще логика обработки

```
def maximum(vs: List[Int]): Int =  
  vs match  
    case Nil ⇒ throw new IndexOutOfBoundsException()  
    case head::rest ⇒ rest.fold(head)(_ max _)
```

// если в языке есть исключения и с ними удобно работать

Например, проще логика обработки

```
def maximum(vs: List[Int]): (Boolean, Int) =  
  vs match  
    case Nil ⇒ (false, 0)  
    case head::rest ⇒ (true, rest.fold(head)(_ max _))
```

Например, проще логика обработки

```
def maximum(vs: List[Int]): (Boolean, Int) =  
  vs match  
    case Nil ⇒ (false, 0)  
    case head::rest ⇒ (true, rest.fold(head)(_ max _))  
  
// всё равно возвращается число, которым можно пользоваться,  
// даже если оно не имеет смысла
```


Например, проще логика обработки

```
enum Option[+A]:                                // из стандартной библиотеки
  case None
  case Some(a: A)

def maximum(vs: List[Int]): (Boolean, Int) =
  vs match
    case Nil ⇒ (false, 0)
    case head::rest ⇒ (true, rest.fold(head)(_ max _))
```

Например, проще логика обработки

```
enum Option[+A]:                                     // из стандартной библиотеки
  case None
  case Some(a: A)

def maximum(vs: List[Int]): Option[Int] =
  vs match
    case Nil => None
    case head::rest => Some(rest.fold(head)(_ max _))
```

Например, проще логика обработки

```
enum Option[+A]:                                // из стандартной библиотеки
  case None
  case Some(a: A)

def maximum(vs: List[Int]): Option[Int] =
  vs match
    case Nil => None
    case head::rest => Some(rest.fold(head)(_ max _))

// спорно? ;-)
```

Например, проще логика обработки

```
enum Option[+A]:                                     // из стандартной библиотеки
  case None
  case Some(a: A)

def maximum(vs: List[Int]): Option[Int] =
  vs match
    case Nil => None
    case head::rest => Some(rest.fold(head)(_ max _))

// спорно? ;-)
```

// я утверждаю, что тут хорошо выражены намерения

Например, проще логика обработки

```
enum Option[+A]:                                // из стандартной библиотеки
  case None
  case Some(a: A)

def maximum(vs: List[Int]): Option[Int] =
  vs match
    case Nil => None
    case head::rest => Some(rest.fold(head)(_ max _))

// спорно? ;-)
```

// я утверждаю, что тут хорошо выражены намерения

// компилятор не даст забыть обработать особый случай

Например, проще логика обработки

```
enum Option[+A]:                                     // из стандартной библиотеки
  case None
  case Some(a: A)

def maximum(vs: List[Int]): Option[Int] =
  vs match
    case Nil => None
    case head::rest => Some(rest.fold(head)(_ max _))

// спорно? ;- )
// я утверждаю, что тут хорошо выражены намерения
// компилятор не даст забыть обработать особый случай
// но можно и лучше
```

Например, меньше ненужных проверок

Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil ⇒ None  
    case head::rest ⇒ Some(rest.fold(head)(_ max _))  
  
final case class Stat(min: Int, max: Int, av: Double)
```


Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil ⇒ None  
    case head::rest ⇒ Some(rest.fold(head)(_ max _))  
  
final case class Stat(min: Int, max: Int, av: Double)  
  
def stat(vs: List[Int]): Option[Stat] =  
  ???
```

Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil ⇒ None  
    case head::rest ⇒ Some(rest.fold(head)(_ max _))  
  
final case class Stat(min: Int, max: Int, av: Double)  
  
def stat(vs: List[Int]): Option[Stat] =  
  vs match  
    case Nil ⇒ None  
    case _::_ ⇒  
      ???
```

Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil ⇒ None  
    case head::rest ⇒ Some(rest.fold(head)(_ max _))
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(vs: List[Int]): Option[Stat] =  
  vs match  
    case Nil ⇒ None  
    case _::_ ⇒  
      maximum(vs) match  
        case None ⇒ ???  
        case Some(ma) ⇒  
          ???
```

Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil ⇒ None  
    case head::rest ⇒ Some(rest.fold(head)(_ max _))  
  
final case class Stat(min: Int, max: Int, av: Double)  
  
def stat(vs: List[Int]): Option[Stat] =  
  vs match  
    case Nil ⇒ None  
    case _::_ ⇒  
      maximum(vs) match  
        case None ⇒ ??? // но мы уже знаем, что сюда не попадём  
        case Some(ma) ⇒  
          ???
```

Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil => None  
    case head::rest => Some(rest.fold(head)(_ max _))  
  
final case class Stat(min: Int, max: Int, av: Double)  
  
def stat(vs: List[Int]): Option[Stat] =  
  maximum(vs) match  
    case None => None  
    case Some(ma) =>  
      ???
```

Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil ⇒ None  
    case head::rest ⇒ Some(rest.fold(head)(_ max _))
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(vs: List[Int]): Option[Stat] =  
  maximum(vs) match  
    case None ⇒ None  
    case Some(ma) ⇒  
      minimum(vs) match  
        case None ⇒ ???  
        case Some(mi) ⇒  
          ???
```

Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil ⇒ None  
    case head::rest ⇒ Some(rest.fold(head)(_ max _))
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(vs: List[Int]): Option[Stat] =  
  maximum(vs) match  
    case None ⇒ None  
    case Some(ma) ⇒  
      minimum(vs) match  
        case None ⇒ ??? // шо, опять?!  
        case Some(mi) ⇒  
          ???
```

Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil ⇒ None  
    case head::rest ⇒ Some(rest.fold(head)(_ max _))  
  
final case class Stat(min: Int, max: Int, av: Double)  
  
def stat(vs: List[Int]): Option[Stat] =  
  for ma ← maximum(vs)  
    mi ← minimum(vs)  
    su ← sum(vs)  
  yield Stat(mi, ma, su / vs.length.toDouble)
```


Например, меньше ненужных проверок

```
def maximum(vs: List[Int]): Option[Int] =  
  vs match  
    case Nil ⇒ None  
    case head::rest ⇒ Some(rest.fold(head)(_ max _))  
  
final case class Stat(min: Int, max: Int, av: Double)  
  
def stat(vs: List[Int]): Option[Stat] =  
  for ma ← maximum(vs) // тут проверка :-)  
    mi ← minimum(vs)   // тут проверка :-|  
    su ← sum(vs)        // тут проверка :-(  
  yield Stat(mi, ma, su / vs.length.toDouble)
```

Например, меньше ненужных проверок

```
def maximum(vs: NonEmptyList[Int]): Int =  
  ???
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(vs: List[Int]): Option[Stat] =  
  for ma ← maximum(vs) // тут проверка :-)  
    mi ← minimum(vs) // тут проверка :-|  
    su ← sum(vs) // тут проверка :-(  
  yield Stat(mi, ma, su / vs.length.toDouble)
```

Например, меньше ненужных проверок

```
def maximum(vs: NonEmptyList[Int]): Int =  
  ???
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(vs: List[Int]): Option[Stat] =  
  vs.toNel match  
    case None ⇒ None  
    case Some(ne) ⇒  
      val ma = maximum(ne)  
      val mi = minimum(ne)  
      val su = sum(ne)  
      Some(Stat(mi, ma, su / ne.length.toDouble))
```

Например, меньше ненужных проверок

```
def maximum(vs: NonEmptyList[Int]): Int =  
  vs.reduceLeft(_ max _)
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(vs: List[Int]): Option[Stat] =  
  vs.toNel match  
    case None ⇒ None  
    case Some(ne) ⇒  
      val ma = maximum(ne)  
      val mi = minimum(ne)  
      val su = sum(ne)  
      Some(Stat(mi, ma, su / ne.length.toDouble))
```

Например, меньше ненужных проверок

```
def maximum(vs: NonEmptyList[Int]): Int =  
  vs.reduceLeft(_ max _)
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(vs: List[Int]): Option[Stat] =  
  vs.toNel.map: ne =>  
    val ma = maximum(ne)  
    val mi = minimum(ne)  
    val su = sum(ne)  
    Stat(mi, ma, su / ne.length.toDouble)
```

Например, меньше ненужных проверок

```
def maximum(vs: NonEmptyList[Int]): Int =  
  vs.reduceLeft(_ max _)
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(ne: NonEmptyList[Int]): Stat =  
  val ma = maximum(ne)  
  val mi = minimum(ne)  
  val su = sum(ne)  
  Stat(mi, ma, su / ne.length.toDouble)
```

Например, меньше ненужных проверок

```
def maximum(vs: NonEmptyList[Int]): Int =  
  vs.reduceLeft(_ max _)
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(ne: NonEmptyList[Int]): Stat =  
  val ma = maximum(ne)  
  val mi = minimum(ne)  
  val su = sum(ne)  
  Stat(mi, ma, su / ne.length.toDouble)
```

```
// Важно: тип `NonEmptyList` объявлен так, чтобы было невозможно  
// создать его значение пустым
```

Например, меньше ненужных проверок

```
def maximum(vs: NonEmptyList[Int]): Int =  
  vs.reduceLeft(_ max _)
```

```
final case class Stat(min: Int, max: Int, av: Double)
```

```
def stat(ne: NonEmptyList[Int]): Stat =  
  val ma = maximum(ne)  
  val mi = minimum(ne)  
  val su = sum(ne)  
  Stat(mi, ma, su / ne.length.toDouble)
```

```
// Важно: тип `NonEmptyList` объявлен так, чтобы было невозможно  
// создать его значение пустым
```


Ещё о намерениях и проверках

```
int main() {  
    int a[5] = {1, 2, 3, 4, 5};  
    for (int i = 0; i < 5; ++i) {  
        printf("a[%d] = %d\n", i, a[i]);  
    }  
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);
```

```
int main() {  
    int a[5] = {1, 2, 3, 4, 5};  
    for (int i = 0; i < 5; ++i) {  
        printf("a[%d] = %d\n", i, a[i]);  
    }  
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);
```

```
int main() {  
    int a[5] = {1, 2, 3, 4, 5};  
    for (int i = start(a); i < 5; ++i) {  
        printf("a[%d] = %d\n", i, a[i]);  
    }  
}
```

Ещё о намерениях и проверках

```
int start(int a[6]);
```

```
int main() {  
    int a[5] = {1, 2, 3, 4, 5};  
    for (int i = start(a); i < 5; ++i) {  
        printf("a[%d] = %d\n", i, a[i]);  
    }  
}
```

Ещё о намерениях и проверках

```
int start(int a[6]);
```

```
a.c: In function 'main':
```

```
a.c:50:16: warning: 'start' accessing 24 bytes in a region of size 20
```

```
50 |   for (int i = start(a); i < 5; ++i) {  
    |               ^~~~~~
```

```
int a[5] = {1, 2, 3, 4, 5};
```

```
for (int i = start(a); i < 5; ++i) {
```

```
    printf("a[%d] = %d\n", i, a[i]);
```

```
}
```

```
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);
```

```
int main() {  
    int a[5] = {1, 2, 3, 4, 5};  
    for (int i = start(a); i < 5; ++i) {  
        printf("a[%d] = %d\n", i, a[i]);  
    }  
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);  
int f(int n, int a[n]) {  
  
    // ...  
  
}  
  
int main() {  
    int a[5] = {1, 2, 3, 4, 5};  
    for (int i = start(a); i < 5; ++i) {  
        printf("a[%d] = %d\n", i, a[i]);  
    }  
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);
```

```
int f(int n, int a[n]) {
```

```
    // ...
```

```
}
```

```
int main() {
```

```
    int a[5] = {1, 2, 3, 4, 5};
```

```
    for (int i = start(a); i < f(5, a); ++i) {
```

```
        printf("a[%d] = %d\n", i, a[i]);
```

```
    }
```

```
}
```


Ещё о намерениях и проверках

```
int start(int a[5]);
```

```
int f(int n, int a[n]) {
```

```
    // ...
```

```
}
```

```
int main() {
```

```
    int a[5] = {1, 2, 3, 4, 5};
```

```
    for (int i = start(a); i < f(6, a); ++i) {
```

```
        printf("a[%d] = %d\n", i, a[i]);
```

```
    }
```

```
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);  
int f(int n, int a[n]) {
```

```
//
```

```
a.c: In function 'main':
```

```
a.c:50:30: warning: 'f' accessing 24 bytes in a region of size 20
```

```
50 |   for (int i = start(a); i < f(6, a); ++i) {  
    |                               ^~~~~~
```

```
int a[5] = {1, 2, 3, 4, 5};  
for (int i = start(a); i < f(6, a); ++i) {  
    printf("a[%d] = %d\n", i, a[i]);  
}
```

```
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);
```

```
int f(int n, int a[n]) {
```

```
    // ...
```

```
}
```

```
int main() {
```

```
    int a[5] = {1, 2, 3, 4, 5};
```

```
    for (int i = start(a); i < f(5, a); ++i) {
```

```
        printf("a[%d] = %d\n", i, a[i]);
```

```
    }
```

```
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);
```

```
int f(int n, int m, int a[n]) {
```

```
// ...
```

```
}
```

```
int main() {
```

```
    int a[5] = {1, 2, 3, 4, 5};
```

```
    for (int i = start(a); i < f(5, i, a); ++i) {
```

```
        printf("a[%d] = %d\n", i, a[i]);
```

```
    }
```

```
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n]) {
    // ...
    if (m ≥ n) return a[n];
    // ...
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    for (int i = start(a); i < f(5, i, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n]) {
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    for (int i = start(a); i < f(5, i, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n]) {
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    int res = a[m];
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    for (int i = start(a); i < f(5, i, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n]) {
    int res = a[m];
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    for (int i = start(a); i < f(5, i, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```


Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n]) {
    int res = a[m]; // ← !!!
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    for (int i = start(a); i < f(5, i, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n + m]) {
    int res = a[m]; // ← !!!
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    for (int i = start(a); i < f(5, i, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n + m]) {
    int res = a[m]; // ← !!!
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5}; // ← !!!
    for (int i = start(a); i < f(5, i, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n + m]) {
    int res = a[m]; // ← !!!
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5}; // ← !!!
    for (int i = start(a); i < f(5, 6, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n + m], int b[start(a)]) {
    int res = a[m]; // ← !!!
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5}; // ← !!!
    for (int i = start(a); i < f(5, 6, a, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Ещё о намерениях и проверках

```
int start(int a[5]);

int f(int n, int m, int a[n + m], int b[start(a)]) {
    int res = a[m]; // ← !!!
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5}; // !!!
    for (int i = start(a); i < f(5, 6, a, a); ++i) { // !!!
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○●

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

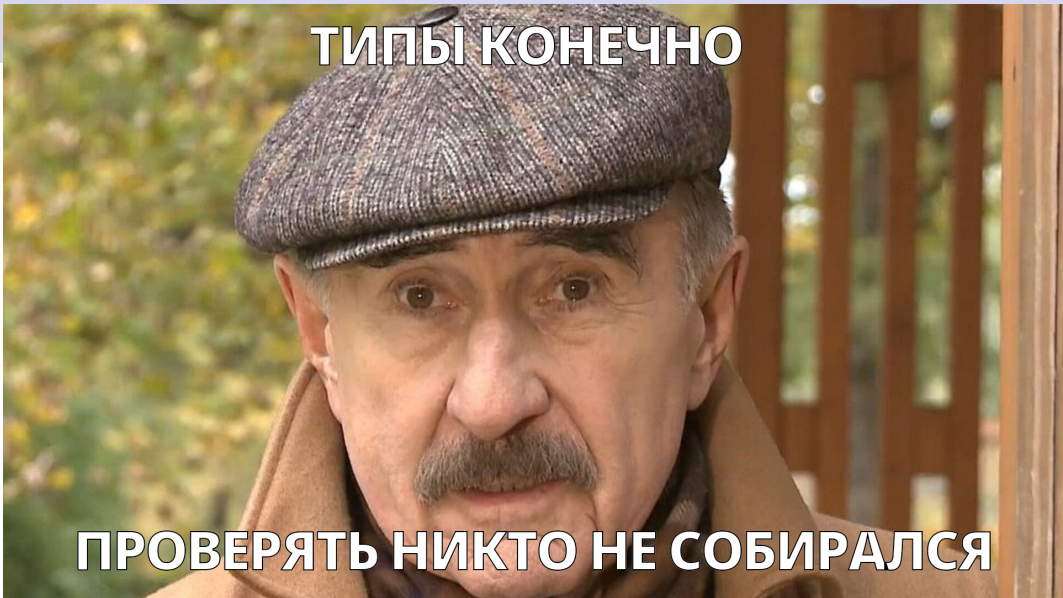
опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

ТИПЫ КОНЕЧНО

ПРОВЕРЯТЬ НИКТО НЕ СОБИРАЛСЯ



Ещё о намерениях и проверках

```
int start(int a[5]) { /* ... */ }

int f(int n, int m, int a[n + m], int b[start(a)]) {
    int res = a[m]; // ← !!!
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5}; // !!!
    for (int i = start(a); i < f(5, 6, a, a); ++i) { // !!!
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```


Ещё о намерениях и проверках

```
int start(int a[5]) { /* ...императивный код... */ }

int f(int n, int m, int a[n + m], int b[start(a)]) {
    int res = a[m]; // ← !!!
    // ...
    if (m ≥ n) return a[n]; // ← !!!
    // ...
    a[m] = 0;
    return res;
}

int main() {
    int a[5] = {1, 2, 3, 4, 5}; //      !!!      !!!
    for (int i = start(a); i < f(5, 6, a, a); ++i) {
        printf("a[%d] = %d\n", i, a[i]);
    }
}
```

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
●○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Что такое *функциональное* программирование?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️
- Что-то заумное?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️
- Что-то заумное?
- неизменяемые структуры данных?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️
- Что-то заумное?
- неизменяемые структуры данных?
- Нет побочных эффектов?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️
- Что-то заумное?
- неизменяемые структуры данных?
- Нет побочных эффектов?
- Функции могут принимать функции и возвращать функции?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️
- Что-то заумное?
- неизменяемые структуры данных?
- Нет побочных эффектов?
- Функции могут принимать функции и возвращать функции?
- Медленное?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️
- Что-то заумное?
- Неизменяемые структуры данных?
- Нет побочных эффектов?
- Функции могут принимать функции и возвращать функции?
- Медленное?
- Требуется много памяти?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️
- Что-то заумное?
- Неизменяемые структуры данных?
- Нет побочных эффектов?
- Функции могут принимать функции и возвращать функции?
- Медленное?
- Требуется много памяти?
- Требуется garbage collector?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️
- Что-то заумное?
- Неизменяемые структуры данных?
- Нет побочных эффектов?
- Функции могут принимать функции и возвращать функции?
- Медленное?
- Требуется много памяти?
- Требуется garbage collector?
- Нет классов, наследования?

Что такое *функциональное* программирование?

- В нём есть функции? ;-)
- Что-то связанное с лямбдами? ...с *монадами*? ☹️
- Что-то заумное?
- неизменяемые структуры данных?
- Нет побочных эффектов?
- Функции могут принимать функции и возвращать функции?
- Медленное?
- Требуется много памяти?
- Требуется garbage collector?
- Нет классов, наследования?
- Прimitives данные?

Единственное общее

- Функции могут принимать функции и возвращать функции!

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
●○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Единственное общее

- Функции высших порядков

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○●○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Все ли функции одинаково полезны?

Языки с функциями высших порядков бывают...

Языки с функциями высших порядков бывают...

- нетипизированные

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные
- слабо типизированные

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные
- слабо типизированные
- объектно-ориентированные

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные
- слабо типизированные
- объектно-ориентированные
- с побочными эффектами

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные
- слабо типизированные
- объектно-ориентированные
- с побочными эффектами
- даже неисполнимые
- ...

Языки с функциями высших порядков бывают...

- нетипизированные
 - динамически типизированные
 - слабо типизированные
 - объектно-ориентированные
 - с побочными эффектами
 - даже неисполнимые
 - ...
- хорошо, что не все они такие ;-)

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные
- слабо типизированные
- объектно-ориентированные
- с побочными эффектами
- даже неисполнимые
- ...
- хорошо, что не все они такие ;-)
- нам сейчас важны только

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные
- слабо типизированные
- объектно-ориентированные
- с побочными эффектами
- даже неисполнимые
- ...
- хорошо, что не все они такие ;-)
- нам сейчас важны только
 - выразительность системы типов

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные
- слабо типизированные
- объектно-ориентированные
- с побочными эффектами
- даже неисполнимые
- ...
- хорошо, что не все они такие ;-)
- нам сейчас важны только
 - выразительность системы типов
 - безопасность, анализируемость

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные
- слабо типизированные
- объектно-ориентированные
- с побочными эффектами
- даже неисполнимые
- ...

- хорошо, что не все они такие ;-)
- нам сейчас важны только
 - выразительность системы типов
 - безопасность, анализируемость
 - как следствие
 - статическая и сильная типизация

Языки с функциями высших порядков бывают...

- нетипизированные
- динамически типизированные
- слабо типизированные
- объектно-ориентированные
- с побочными эффектами
- даже неисполнимые
- ...

- хорошо, что не все они такие ;-)
- нам сейчас важны только
 - выразительность системы типов
 - безопасность, анализируемость
 - как следствие
 - статическая и сильная типизация
 - чистота^a, ссылочная прозрачность^b

^apure functional programming

^breferential transparency

Ссылочная прозрачность

```
someVal : Nat
someVal = 5
```

Ссылочная прозрачность

```
someVal : Nat  
someVal = 5
```

```
someFun : Nat → Nat  
someFun x = product [1..someVal] + x
```


Ссылочная прозрачность

```
someVal : Nat
someVal = 5
```

```
someFun : Nat → Nat
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()
anotherFun x y = do
  let n = someFun x
  let m = someFun y
  let xs = List.replicate (n + n) m
  println xs
```

Ссылочная прозрачность¹

```
someVal : Nat  
someVal = 5
```

```
someFun : Nat → Nat  
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()  
anotherFun x y = do  
  let n = someFun x  
  let m = someFun y  
  let xs = List.replicate (n + n) m  
  println xs
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Ссылочная прозрачность¹

```
someVal : Nat
someVal = 5
```

```
someFun : Nat → Nat
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()
anotherFun x y = do
  let n = someFun x
  let m = someFun y
  println $ List.replicate (n + n) m
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Ссылочная прозрачность¹

```
someVal : Nat  
someVal = 5
```

```
someFun : Nat → Nat  
someFun x = product [1..5] + x
```

```
anotherFun : Nat → Nat → IO ()  
anotherFun x y = do  
  let n = someFun x  
  let m = someFun y  
  println $ List.replicate (n + n) m
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Ссылочная прозрачность¹

```
someVal : Nat
someVal = 5
```

```
someFun : Nat → Nat
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()
anotherFun x y = do
  let n = someFun x
  let m = someFun y
  println $ List.replicate (n + n) m
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Ссылочная прозрачность¹

```
someVal : Nat  
someVal = 5
```

```
someFun : Nat → Nat  
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()  
anotherFun x y = do  
  —let n = someFun x  
  let m = someFun y  
  println $ List.replicate (someFun x + someFun x) m
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Ссылочная прозрачность¹

```
someVal : Nat
someVal = 5
```

```
someFun : Nat → Nat
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()
anotherFun x y = do
  —let n = someFun x
  let m = someFun y
  println $ List.replicate (someFun x + someFun x) m
  — три вызова `someFun` Вместо двух! 0_0
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Ссылочная прозрачность¹

```
someVal : Nat
someVal = 5
```

```
someFun : Nat → Nat
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()
anotherFun x y = do
  —let n = someFun x
  let m = someFun y
  println $ List.replicate (someFun x + someFun x) m
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Ссылочная прозрачность¹

```
someVal : Nat  
someVal = 5
```

```
someFun : Nat → Nat  
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()  
anotherFun x y = do  
  —let n = someFun x  
  let m = product [1..5] + y  
  println $  
    List.replicate ((product [1..someVal] + x) + someFun x) m
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Ссылочная прозрачность¹

```
someVal : Nat
someVal = 5
```

```
someFun : Nat → Nat
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()
anotherFun x y = do
  —let n = someFun x
  let m = product [1..5] + y
  println $
    List.replicate ((product [1..5] + x) + (product [1..5] + x)) m
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Ссылочная прозрачность¹

```
someVal : Nat
someVal = 5
```

```
someFun : Nat → Nat
someFun x = product [1..someVal] + x
```

```
anotherFun : Nat → Nat → IO ()
anotherFun x y = do
  let cse0 = product [1..5]
  let cse1 = cse0 + x
  println $ List.replicate (cse1 + cse1) (cse0 + y)
```

¹ Любое вхождение *любой* переменной можно заменить на её значение без влияния на результат

Проверки и безопасность

`index : List a → (idx : Nat) → a`

Проверки и безопасность

`index : List a → (idx : Nat) → Maybe a`

Проверки и безопасность

```
index : List a → ((idx : Nat) → Maybe a)
```

Проверки и безопасность

`index : List a → (idx : Nat) → Maybe a`

Проверки и безопасность

```
index : List a → (idx : Fin ?len) → a
```


Проверки и безопасность

```
index : (xs : List a) → (idx : Fin ?len) → a
```

Проверки и безопасность

`index : (xs : List a) → (idx : Fin (length xs)) → a`

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
```

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
```

```
fifth = index [1, 2, 3, 4, 5] 5
```

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
```

```
fifth = index [1, 2, 3, 4, 5] 5
```

```
Error: While processing right hand side of y. Can't find an implementation
for So (integerLessThanNat 5 (length [1, 2, 3, 4, 5])).
```

```
Main:264:27—264:28
```

```
260 | index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
261 |
```

```
262 | third = index [1, 2, 3, 4, 5] 3
```

```
263 | fifth = index [1, 2, 3, 4, 5] 5
```

```
^
```

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
```

```
fifth = index [1, 2, 3, 4, 5] 5
```

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3  
failing "Can't find an implementation"  
fifth = index [1, 2, 3, 4, 5] 5
```

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3  
failing "Can't find an implementation"  
fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
```


Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else ?go_forward
```

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else findFrom xs elem (FS idx)
```

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
```

```
failing "Can't find an implementation"
```

```
Error: While processing right hand side of findFrom.  
Can't solve constraint between: S (length xs) and length xs.
```

```
Main:352:28—352:34
```

```
348 | Int → (idx : Fin (length xs)) → Bool  
349 | findFrom xs elem idx =  
350 |   if index xs idx == elem  
351 |   then True  
352 |   else findFrom xs elem (FS idx)  
      ^^^^^^
```

Bool

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else findFrom xs elem (FS idx)
```

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else let idx' = FS idx in
       ?with_next_idx
```

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
xs : List Int
idx : Fin (length xs)
elem : Int
idx' : Fin (S (length xs))
```

```
have_next_idx : Bool
```

```
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else let idx' = FS idx in
       ?with_next_idx
```

Bool

Проверки и безопасность

```
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else let idx' = FS idx in
       ?with_next_idx
```

Проверки и безопасность

```
strengthen : Fin (S n) → Maybe (Fin n)
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
  fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else let idx' = FS idx in
       ?with_next_idx
```


Проверки и безопасность

```
strengthen : Fin (S n) → Maybe (Fin n)
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
  fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else case strengthen (FS idx) of
    Nothing   ⇒ False
    Just idx' ⇒ ?with_next_idx
```

Проверки и безопасность

```
strengthen : Fin (S n) → Maybe (Fin n)
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
xs : List Int
idx : Fin (length xs)
elem : Int
idx' : Fin (length xs)
```

```
have_next_idx : Bool
```

```
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else case strengthen (FS idx) of
    Nothing   ⇒ False
    Just idx' ⇒ ?with_next_idx
```

Bool

Проверки и безопасность

```
strengthen : Fin (S n) → Maybe (Fin n)
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
  fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else case strengthen (FS idx) of
    Nothing   ⇒ False
    Just idx' ⇒ ?with_next_idx
```

Проверки и безопасность

```
strengthen : Fin (S n) → Maybe (Fin n)
index : (xs : List a) → (idx : Fin (length xs)) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
  fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : (xs : List Int) → Int → (idx : Fin (length xs)) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else case strengthen (FS idx) of
    Nothing   ⇒ False
    Just idx' ⇒ findFrom xs elem idx'
```

Проверки и безопасность

```
strengthen : Fin (S n) → Maybe (Fin n)
index : Vect n a → (idx : Fin n) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
  fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : {n : _} → Vect n Int → Int → (idx : Fin n) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else case strengthen (FS idx) of
    Nothing   ⇒ False
    Just idx' ⇒ findFrom xs elem idx'
```

Проверки и безопасность

```
strengthen : Fin (S n) → Maybe (Fin n)
index : Vect n a → (idx : Fin n) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : {n : _} → Vect n Int → Int → (idx : Fin n) → Bool
findFrom xs elem idx =
  if index xs idx == elem
  then True
  else case strengthen (FS idx) of
    Nothing   ⇒ False
    Just idx' ⇒ findFrom xs elem idx'
```

—
—
—
—
—

↖
← WTF???

Проверки и безопасность

```
strengthen : Fin (S n) → Maybe (Fin n)
index : Vect n a → (idx : Fin n) → a
```

```
third = index [1, 2, 3, 4, 5] 3
failing "Can't find an implementation"
  fifth = index [1, 2, 3, 4, 5] 5
```

```
findFrom : Vect n Int → Int → (idx : Fin n) → Bool
findFrom xs      elem FZ      = any (== elem) xs
findFrom (_::xs) elem (FS i) = findFrom xs elem i
```

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○●○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Больше намерений

```
data Vect : (length : Nat) → Type → Type
```


Больше намерений

```
data Vect : (length : Nat) → Type → Type
```

```
data BalancedTree : (depth : Nat) → Type → Type where
```

Больше намерений

```
data Vect : (length : Nat) → Type → Type
```

```
data BalancedTree : (depth : Nat) → Type → Type where  
  Leaf : a → BalancedTree 1 a
```

Больше намерений

```
data Vect : (length : Nat) → Type → Type
```

```
data BalancedTree : (depth : Nat) → Type → Type where  
  Leaf : a → BalancedTree 1 a  
  Node : (left, right : BalancedTree n a) → BalancedTree (1 + n) a
```

Больше намерений

```
data Vect : (length : Nat) → Type → Type
```

```
data BalancedTree : (depth : Nat) → Type → Type where  
  Leaf : a → BalancedTree 1 a  
  Node : (left, right : BalancedTree n a) → BalancedTree (1 + n) a
```

```
data IntsTree : (minD, maxD : Nat) → (sum : Int) → Type where
```

Больше намерений

```
data Vect : (length : Nat) → Type → Type
```

```
data BalancedTree : (depth : Nat) → Type → Type where
  Leaf : a → BalancedTree 1 a
  Node : (left, right : BalancedTree n a) → BalancedTree (1 + n) a
```

```
data IntsTree : (minD, maxD : Nat) → (sum : Int) → Type where
  Leaf : (n : Int) → IntsTree 1 1 n
```

Больше намерений

```
data Vect : (length : Nat) → Type → Type
```

```
data BalancedTree : (depth : Nat) → Type → Type where  
  Leaf : a → BalancedTree 1 a  
  Node : (left, right : BalancedTree n a) → BalancedTree (1 + n) a
```

```
data IntsTree : (minD, maxD : Nat) → (sum : Int) → Type where  
  Leaf : (n : Int) → IntsTree 1 1 n  
  Node : (left : IntsTree ldmi ldma ls) →  
         (right : IntsTree rdmi rdma rs) →
```

Больше намерений

```
data Vect : (length : Nat) → Type → Type
```

```
data BalancedTree : (depth : Nat) → Type → Type where
  Leaf : a → BalancedTree 1 a
  Node : (left, right : BalancedTree n a) → BalancedTree (1 + n) a
```

```
data IntsTree : (minD, maxD : Nat) → (sum : Int) → Type where
  Leaf : (n : Int) → IntsTree 1 1 n
  Node : (left : IntsTree ldmi ldma ls) →
    (right : IntsTree rdmi rdma rs) →
    IntsTree (ldmi `min` rdmi) (ldma `max` rdma) (n + m)
```

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○●○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Рассуждать про списки?

Рассуждать про списки?

```
data Has : List a → a → Type where
```

Рассуждать про списки?

```
data Has : List a → a → Type where
  Here : x :: xs → Has x
```

Рассуждать про списки?

```
data Has : List a → a → Type where
  Here  : x :: xs `Has` x
  There : xs `Has` e → x :: xs `Has` e
```

Рассуждать про списки?

```
data Has : List a → a → Type where
  Here   : x :: xs `Has` x
  There  : xs `Has` e → x :: xs `Has` e
```

```
data ListWith5 : Type where
  With5 : (xs : List Nat) → xs `Has` 5 ⇒ ListWith5
```

Рассуждать про списки?

```
data Has : List a → a → Type where
  Here   : x :: xs `Has` x
  There  : xs `Has` e → x :: xs `Has` e
```

```
data ListWith5 : Type where
  With5 : (xs : List Nat) → xs `Has` 5 ⇒ ListWith5
```

```
good = With5 [1, 2, 3, 4, 5]
```

Рассуждать про списки?

```
data Has : List a → a → Type where
  Here  : x :: xs `Has` x
  There : xs `Has` e → x :: xs `Has` e
```

```
data ListWith5 : Type where
  With5 : (xs : List Nat) → xs `Has` 5 ⇒ ListWith5
```

```
good = With5 [1, 2, 3, 4, 5]
```

```
failing "Can't find an implementation for Has [1, 2, 3, 4, 6] 5"
bad = With5 [1, 2, 3, 4, 6]
```

Списки

```
data NList : Type where
  Nil      : NList
  (::)     : Nat → NList → NList
```

Списки

```
data NList : Type where
  Nil      : NList
  (::)     : Nat → NList → NList
```

```
actuallySorted : NList
actuallySorted = 1 :: 2 :: 5 :: 9 :: Nil
```


Списки

```
data NList : Type where
  Nil      : NList
  (::)     : Nat → NList → NList
```

```
actuallySorted : NList
actuallySorted = 1 :: 2 :: 5 :: 9 :: Nil
```

```
unsorted : NList
unsorted = 1 :: 5 :: 2 :: 9 :: 1 :: Nil
```

Списки сортированные?

```
data NList : Type where
  Nil      : NList
  (::)     : Nat → NList → NList
```

Списки сортированные?

```
data SortNList : Type where
  Nil  : SortNList
  (::) : Nat → SortNList → SortNList
```

Списки сортированные?

```
data SortNList : Type where
  Nil      : SortNList
  (::) : Nat → SortNList → SortNList
    — голова субсписка (если есть) должна быть больше числа
```

Списки сортированные?

```
data SortNList : Type where
  Nil      : SortNList
  (::) : (x : Nat) → (xs : SortNList) → SortNList
      — голова субсписка (если есть) должна быть больше числа
```

Списки сортированные?

```
data SortNList : Type where
  Nil      : SortNList
  (::)      : (x : Nat) → (xs : SortNList) → SortNList
            — голова `xs` (если есть) должна быть больше `x`
```

Списки сортированные!

```
data SortNList : Type where
  Nil  : SortNList
  (::)  : (x : Nat) → (xs : SortNList) → SortNList
        — голова `xs` (если есть) должна быть больше `x`

data HeadGT : Nat → SortNList → Type where
  E  : HeadGT n []
  NE : x `GT` n → HeadGT n (x::xs)
```

Списки сортированные!

```
data SortNList : Type where
  Nil  : SortNList
  (::)  : (x : Nat) → (xs : SortNList) → HeadGT x xs ⇒ SortNList
      — голова `xs` (если есть) должна быть больше `x`

data HeadGT : Nat → SortNList → Type where
  E  : HeadGT n []
  NE : x `GT` n → HeadGT n $ (x::xs) @{prf}
```


Списки сортированные!

```
data SortNList : Type where
  Nil  : SortNList
  (::)  : (x : Nat) → (xs : SortNList) → HeadGT x xs ⇒ SortNList
```

```
data HeadGT : Nat → SortNList → Type where
  E  : HeadGT n []
  NE : x `GT` n → HeadGT n $ (x::xs) @{prf}
```

Списки сортированные!

```
data SortNList : Type where
  Nil  : SortNList
  (::)  : (x : Nat) → (xs : SortNList) → HeadGT x xs ⇒ SortNList
```

```
data HeadGT : Nat → SortNList → Type where
  E  : HeadGT n []
  NE : x `GT` n → HeadGT n $ (x::xs) @{prf}
```

```
actuallySorted : SortNList
actuallySorted = 1 :: 2 :: 5 :: 9 :: Nil
```

Списки сортированные!

```
data SortNList : Type where
  Nil  : SortNList
  (::)  : (x : Nat) → (xs : SortNList) → HeadGT x xs ⇒ SortNList
```

```
data HeadGT : Nat → SortNList → Type where
  E  : HeadGT n []
  NE : x `GT` n → HeadGT n $ (x::xs) @{prf}
```

```
actuallySorted : SortNList
actuallySorted = 1 :: 2 :: 5 :: 9 :: Nil
```

```
failing "Can't find an implementation for HeadGT"
unsorted : SortNList
unsorted = 1 :: 5 :: 2 :: 9 :: 1 :: Nil
```

Списки сортированные!

```
data SortNList : Type where
  Nil  : SortNList
  (::)  : (x : Nat) → (xs : SortNList) → HeadGT x xs ⇒ SortNList
```

```
data HeadGT : Nat → SortNList → Type where
  E  : HeadGT n []
  NE : x `GT` n → HeadGT n $ (x::xs) @{prf}
```

```
arbitrarySortNList : Gen SortNList
```

Списки сортированные!

```
data SortNList : Type where
  Nil  : SortNList
  (::) : (x : Nat) → (xs : SortNList) → HeadGT x xs ⇒ SortNList
```

```
data HeadGT : Nat → SortNList → Type where
  E  : HeadGT n []
  NE : x `GT` n → HeadGT n $ (x::xs) @{prf}
```

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
```

Списки сортированные!

```
data SortNList : Type where
  Nil  : SortNList
  (::)  : (x : Nat) → (xs : SortNList) → HeadGT x xs ⇒ SortNList
```

```
data HeadGT : Nat → SortNList → Type where
  E  : HeadGT n []
  NE : x `GT` n → HeadGT n $ (x::xs) @{prf}
```

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
arbitrarySortNList = deriveGen
```

Списки сортированные!

```
data SortNList : Type where
  Nil      : SortNList
  (::)      : (x : Nat) → (xs : SortNList) → HeadGT x xs ⇒ SortNList
```

```
data HeadGT : Nat → SortNList → Type where
  E      : HeadGT n []
  NE     : x `GT` n → HeadGT n $ (x::xs) @{prf}
```

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
arbitrarySortNList = deriveGen
```

```
—      ^^^^^^^^
—      \
—      ——— DepTyCheck
```

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
●○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Соединяем всё вместе

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
●○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Соединяем всё вместе

```
data SortNList : Type
```

Соединяем всё вместе

```
data SortNList : Type
```

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList  
arbitrarySortNList = deriveGen
```

Соединяем всё вместе

```
data SortNList : Type
```

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList  
arbitrarySortNList = deriveGen
```

```
toList : SortNList → List Nat
```

Соединяем всё вместе

```
data SortNList : Type
```

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList  
arbitrarySortNList = deriveGen
```

```
toList : SortNList → List Nat
```

```
insertOk : Property  
insertOk = property $ \f1 ⇒ do  
  x ← forAll arbitraryNat  
  xs ← forAll $ toList <$> sortedNatList f1  
  assert $ sorted $ insert x xs  
  assert $ x `elem` insert x xs
```

Соединяем всё вместе

— Декларативная спецификация модельных входных данных

```
data SortNList : Type
```

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
```

```
arbitrarySortNList = deriveGen
```

```
toList : SortNList → List Nat
```

```
insertOk : Property
```

```
insertOk = property $ \f1 ⇒ do
```

```
  x ← forAll arbitraryNat
```

```
  xs ← forAll $ toList <$> sortedNatList f1
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```

Соединяем всё вместе

— Декларативная спецификация модельных входных данных

```
data SortNList : Type
```

— Сдериивированный полный генератор

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
```

```
arbitrarySortNList = deriveGen
```

```
toList : SortNList → List Nat
```

```
insertOk : Property
```

```
insertOk = property $ \f1 ⇒ do
```

```
  x ← forAll arbitraryNat
```

```
  xs ← forAll $ toList <$> sortedNatList f1
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```

Соединяем всё вместе

— Декларативная спецификация модельных входных данных

```
data SortNList : Type
```

— Сдериивированный полный генератор + хорошие распределения

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
```

```
arbitrarySortNList = deriveGen
```

```
toList : SortNList → List Nat
```

```
insertOk : Property
```

```
insertOk = property $ \f1 ⇒ do
```

```
  x ← forAll arbitraryNat
```

```
  xs ← forAll $ toList <$> sortedNatList f1
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```

Соединяем всё вместе

— Декларативная спецификация модельных входных данных

```
data SortNList : Type
```

— Сдериивированный полный генератор + хорошие распределения

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
```

```
arbitrarySortNList = deriveGen
```

— Преобразователь в целевые входные данные

```
toList : SortNList → List Nat
```

```
insertOk : Property
```

```
insertOk = property $ \f1 ⇒ do
```

```
  x ← forAll arbitraryNat
```

```
  xs ← forAll $ toList <$> sortedNatList f1
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```


Соединяем всё вместе

— Декларативная спецификация модельных входных данных

```
data SortNList : Type
```

— Сдериивированный полный генератор + хорошие распределения

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
```

```
arbitrarySortNList = deriveGen
```

— Преобразователь в целевые входные данные

```
toList : SortNList → List Nat
```

— Тестируемое свойство

```
insertOk : Property
```

```
insertOk = property $ \f1 ⇒ do
```

```
  x ← forAll arbitraryNat
```

```
  xs ← forAll $ toList <$> sortedNatList f1
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```

Соединяем всё вместе

— Декларативная спецификация (не обязательно полная!)

```
data SortNList : Type
```

— Сдериивированный полный генератор + хорошие распределения

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
```

```
arbitrarySortNList = deriveGen
```

— Преобразователь в целевые входные данные

```
toList : SortNList → List Nat
```

— Тестируемое свойство

```
insertOk : Property
```

```
insertOk = property $ \f1 ⇒ do
```

```
  x ← forAll arbitraryNat
```

```
  xs ← forAll $ toList <$> sortedNatList f1
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```

Соединяем всё вместе

— Декларативная спецификация (не обязательно полная!)

```
data SortNList : Type
```

— Сдериивированный полный генератор + хорошие распределения

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
```

```
arbitrarySortNList = deriveGen
```

— Преобразователь в целевые входные данные

```
toList : SortNList → List Nat
```

— Тестируемое свойство (не обязательно полное!)

```
insertOk : Property
```

```
insertOk = property $ \f1 ⇒ do
```

```
  x ← forAll arbitraryNat
```

```
  xs ← forAll $ toList <$> sortedNatList f1
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```

Dependent types-driven property-based testing

— Декларативная спецификация (не обязательно полная!)

```
data SortNList : Type
```

— Сдериивированный полный генератор + хорошие распределения

```
arbitrarySortNList : Fuel → Gen MaybeEmpty SortNList
```

```
arbitrarySortNList = deriveGen
```

— Преобразователь в целевые входные данные

```
toList : SortNList → List Nat
```

— Тестируемое свойство (не обязательно полное!)

```
insertOk : Property
```

```
insertOk = property $ \f1 ⇒ do
```

```
  x ← forAll arbitraryNat
```

```
  xs ← forAll $ toList <$> sortedNatList f1
```

```
  assert $ sorted $ insert x xs
```

```
  assert $ x `elem` insert x xs
```

Слева
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
●○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

— Декл

data So

— Сдер

arbitra

arbitra

— Прео

toList

— Тест

insert0

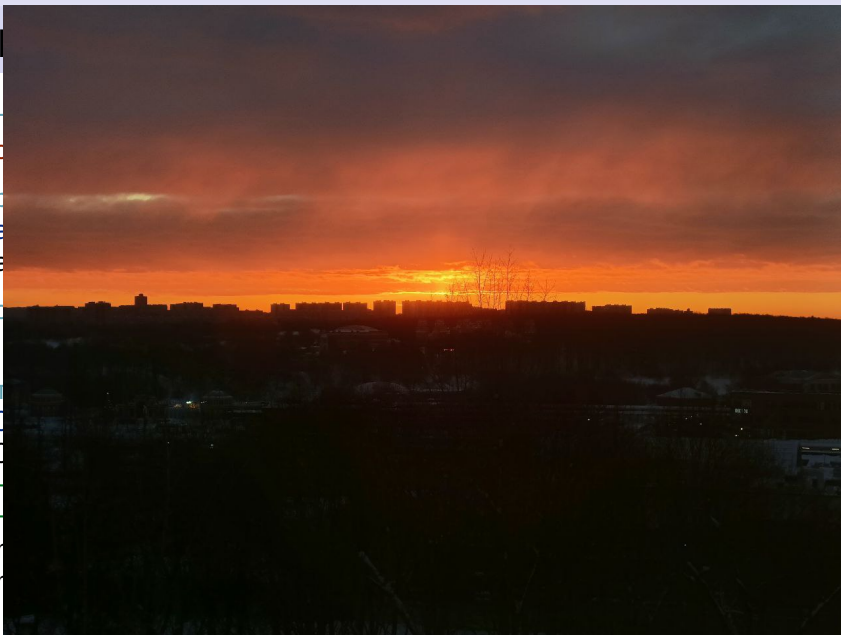
insert0

x ←

xs ←

asser

asser



Я

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○●

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Когда стоит?

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○●

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Когда стоит?

- Описать проще, чем сгенерировать

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○●

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Когда стоит?

- Описать проще, чем сгенерировать
- Входные данные со сложным *инвариантом*

Когда стоит?

- Описать проще, чем сгенерировать
- Входные данные со сложным *инвариантом*
- Входных данных безумно много

Когда стоит?

- Описать проще, чем сгенерировать
- Входные данные со сложным *инвариантом*
- Входных данных безумно много
- Очень много (относительно) независимых сочетаний

Ну, например?

- Описать проще, чем сгенерировать
- Входные данные со сложным *инвариантом*
- Входных данных безумно много
- Очень много (относительно) независимых сочетаний

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
●○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Ну, например

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
●○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Ну, например

- *Одна крупная IT компания с Востока...*

Ну, например

- *Одна крупная IT компания с Востока...*
- Диалект Typescript со статической проверкой типов

Ну, например

- *Одна крупная IT компания с Востока...*
- Диалект Typescript со статической проверкой типов
- Имеет интерпретатор с JIT и компилятор

Ну, например

- *Одна крупная IT компания с Востока...*
- Диалект Typescript со статической проверкой типов
- Имеет интерпретатор с JIT и компилятор
- Свойства

Ну, например

- *Одна крупная IT компания с Востока...*
- Диалект Typescript со статической проверкой типов
- Имеет интерпретатор с JIT и компилятор
- Свойства
 - семантически корректные программы должны компилироваться

Ну, например

- *Одна крупная IT компания с Востока...*
- Диалект Typescript со статической проверкой типов
- Имеет интерпретатор с JIT и компилятор
- Свойства
 - семантически корректные программы должны компилироваться
 - среди них завершающиеся программы должны запускаться без неожиданных падений и зацикливаний

Ну, например

- *Одна крупная IT компания с Востока...*
- Диалект Typescript со статической проверкой типов
- Имеет интерпретатор с JIT и компилятор
- Свойства
 - семантически корректные программы должны компилироваться
 - среди них завершающиеся программы должны запускаться без неожиданных падений и зацикливаний
 - все варианты запуска должны выдавать одинаковый результат

Как это можно специфицировать

```
data Stmts : (functions : List (Name, FunSig)) →  
              (varsBefore : List (Name, Type)) →  
              (varsAfter  : 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 (found lk)) →  
       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
```

Как это можно специфицировать

```
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 (found lk)
```

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

Заметим ортогональность, относительную независимость

```
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 (found lk)
```

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

Семантически корректные программы

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

Семантически корректные программы

```
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]
```


Семантически корректные программы

```
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]])
```

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○●○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Семантически некорректные программы

Семантически некорректные программы

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

Семантически некорректные программы

```
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"]
```

Семантически некорректные программы

```
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"]
```

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○●○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Примéним

Testing...

Примéним

```
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()
...
```

Применим

```
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...

Применим

```
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  
}
```

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○●○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Примéним

Testing...

Примéним

```
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(...)
...
```

Применим

```
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...

Применим

```
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))  
  }  
}
```

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○●

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○○○

Примéним

- ...и так далее

Примéним

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

Примéним

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

Примéним

- ...и так далее
- Было найдено 9 подобных ошибок
 - В JIT-, AOT-оптимизаторе, тайпчекере, кодогенераторе
- Ещё 8 во время написания спецификации
- Специфицировано подмножество
 - Завершающиеся программы
 - Циклы, ветвления, присваивания, исключения
 - Классы без методов, числа, массивы

Примéним

- ...и так далее
- Было найдено 9 подобных ошибок
 - В JIT-, AOT-оптимизаторе, тайпчекере, кодогенераторе
- Ещё 8 во время написания спецификации
- Специфицировано подмножество
 - Завершающиеся программы
 - Циклы, ветвления, присваивания, исключения
 - Классы без методов, числа, массивы

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
●○

и ещё раз опыт
○○○○○

Напоследок
○○○

Ну, например

Ну, например

- Целевая система — драйвер FAT32

Ну, например

- Целевая система — драйвер FAT32
- Семантически корректный образ ФС

Ну, например

- Целевая система — драйвер FAT32
- Семантически корректный образ ФС
- Свойства

Ну, например

- Целевая система — драйвер FAT32
- Семантически корректный образ ФС
- Свойства
 - Любой семантически корректный образ успешно монтируется

Ну, например

- Целевая система — драйвер FAT32
- Семантически корректный образ ФС
- Свойства
 - Любой семантически корректный образ успешно монтируется
 - На примонтированном образе успешно выполняется `ls`

Ну, например

- Целевая система — драйвер FAT32
- Семантически корректный образ ФС
- Свойства
 - Любой семантически корректный образ успешно монтируется
 - На примонтированном образе успешно выполняется `ls`
 - ...и другие допустимые системные вызовы

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○●

и ещё раз опыт
○○○○○

Напоследок
○○○

Примéним

Примéним

- Было найдено три реализации FAT32, не соответствующие спецификации

Примéним

- Было найдено три реализации FAT32, не соответствующие спецификации
- Какие?

Примéним

- Было найдено три реализации FAT32, не соответствующие спецификации
- Какие?

credit: Илья Денисьев

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
●○○○○

Напоследок
○○○

Ну, например

Ну, например

- Целевая система — САПР, поддерживающий SystemVerilog

Ну, например

- Целевая система — САПР, поддерживающий SystemVerilog
- Множество open-source реализаций

Ну, например

- Целевая система — САПР, поддерживающий SystemVerilog
- Множество open-source реализаций
- Семантически корректные определения на SystemVerilog

Ну, например

- Целевая система — САПР, поддерживающий SystemVerilog
- Множество open-source реализаций
- Семантически корректные определения на SystemVerilog
- Свойства

Ну, например

- Целевая система — САПР, поддерживающий SystemVerilog
- Множество open-source реализаций
- Семантически корректные определения на SystemVerilog
- Свойства
 - Любое семантически корректное определения должно успешно приниматься инструментом

Ну, например

- Целевая система — САПР, поддерживающий SystemVerilog
- Множество open-source реализаций
- Семантически корректные определения на SystemVerilog
- Свойства
 - Любое семантически корректное определения должно успешно приниматься инструментом
 - Для инструментов симуляции, любое семантически корректное определение должно успешно симулироваться на заданном ограничении по тактам

Ну, например

- Целевая система — САПР, поддерживающий SystemVerilog
- Множество open-source реализаций
- Семантически корректные определения на SystemVerilog
- Свойства
 - Любое семантически корректное определения должно успешно приниматься инструментом
 - Для инструментов симуляции, любое семантически корректное определение должно успешно симулироваться на заданном ограничении по тактам
- Подмножество
 - Модули, порты, соединения
 - Типы соединений/портов

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○●○○○

Напоследок
○○○

Примéним

Testing...

Применим

iverilog

```
ivl: t-dll-api.cc:2501: ivl_nexus_s* ivl_signal_nex(ivl_signal_t, unsigned int)
Assertion `net→type_ == IVL_SIT_REG' failed.
Aborted
```

¹<https://github.com/steveicarus/iverilog>

Применим

iverilog

```
ivl: t-dll-api.cc:2501: ivl_nexus_s* ivl_signal_nex(ivl_signal_t, unsigned int)
Assertion `net→type_ == IVL_SIT_REG' failed.
Aborted
```

Shrinking...

¹<https://github.com/steveicarus/iverilog>

Применим

```
module a(output uwire o1 [0:1]);  
endmodule
```

²https://deptycheck.github.io/verilog-model/error/t_dll_api_cc_ivl_nexus_s

³<https://github.com/steveicarus/iverilog/issues/1213>

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○●○○

Напоследок
○○○

Примéним

Testing...

Примéним

iverilog

```
error: Array i1 needs an array index here.  
error: Unable to elaborate r-value: i1
```

¹<https://github.com/steveicarus/iverilog>

Примéним

iverilog

```
error: Array i1 needs an array index here.  
error: Unable to elaborate r-value: i1
```

Shrinking...

¹<https://github.com/steveicarus/iverilog>

Применим

```
module a(output o1 [0:0], input i1 [0:0]);  
  assign o1 = i1;  
endmodule: a
```

²https://deptycheck.github.io/verilog-model/error/array_needs_an_array_index_here

³<https://github.com/steveicarus/iverilog/issues/1265>

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○●○

Напоследок
○○○

Примéним

Testing...

Применим

verilator

```
%Error-UNSUPPORTED: test.sv:7:13:
Unsupported tristate port expression: VARREF '__Vcellinpt__a_inst__a1'
note: In instance 'b'
    7 |   a a_inst(.a1(b1));
      |           ^~
... For error description see https://verilator.org/warn/UNSUPPORTED?v=5.039

%Error: Internal Error: test.sv:7:13:
../V3DfgSynthesize.cpp:442: Mismatched width reached DFG
    7 |   a a_inst(.a1(b1));
      |           ^~
... This fatal error may be caused by the earlier error(s); resolve those first
```

¹<https://github.com/verilator/verilator>

Применим

verilator

```
%Error-UNSUPPORTED: test.sv:7:13:
Unsupported tristate port expression: VARREF '__Vcellinpt__a_inst__a1'
note: In instance 'b'
    7 |   a a_inst(.a1(b1));
      |           ^~
... For error description see https://verilator.org/warn/UNSUPPORTED?v=5.039

%Error: Internal Error: test.sv:7:13:
../V3DfgSynthesize.cpp:442: Mismatched width reached DFG
    7 |   a a_inst(.a1(b1));
      |           ^~
... This fatal error may be caused by the earlier error(s); resolve those first
```

Shrinking...

¹<https://github.com/verilator/verilator>

Применим

```
module a(output logic [1:2] a1);  
    assign a1 = 'bz;  
endmodule: a  
  
module b (output logic b1);  
    a a_inst(.a1(b1));  
endmodule: b
```

²https://deptycheck.github.io/verilog-model/error/v3dfgsynthesize_cpp_mismatched_width_reached_dfg

³<https://github.com/verilator/verilator/issues/6323>

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○●

Напоследок
○○○

Применим

- ...и так далее

Применим

- ...и так далее
- 6 инструментов

Применим

- ...и так далее
- 6 инструментов
- Всего найдено 64 несоответствия спецификации

Примéним

- ...и так далее
- 6 инструментов
- Всего найдено 64 несоответствия спецификации
 - признанные новые баги

Примéним

- ...и так далее
- 6 инструментов
- Всего найдено 64 несоответствия спецификации
 - признанные новые баги
 - известные баги

Примéним

- ...и так далее
- 6 инструментов
- Всего найдено 64 несоответствия спецификации
 - признанные новые баги
 - известные баги
 - недореализованности

Применим

- ...и так далее
- 6 инструментов
- Всего найдено 64 несоответствия спецификации
 - признанные новые баги
 - известные баги
 - недореализованности
 - плохие сообщения об ошибках

Применим

- ...и так далее
- 6 инструментов
- Всего найдено 64 несоответствия спецификации
 - признанные новые баги
 - известные баги
 - недореализованности
 - плохие сообщения об ошибках
 - недокументированные особенности

Применим

- ...и так далее
- 6 инструментов
- Всего найдено 64 несоответствия спецификации
 - признанные новые баги
 - известные баги
 - недореализованности
 - плохие сообщения об ошибках
 - недокументированные особенности
- <https://deptycheck.github.io/verilog-model/>

Применим

- ...и так далее
- 6 инструментов
- Всего найдено 64 несоответствия спецификации
 - признанные новые баги
 - известные баги
 - недореализованности
 - плохие сообщения об ошибках
 - недокументированные особенности
- <https://deptycheck.github.io/verilog-model/>

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
●○○

Может возникнуть вопрос

Может возникнуть вопрос

- Зачем зависимые типы?

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!
- Но использования типов данных

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!
- Но использования типов данных
 - облегчает рассуждение о генерации входных данных

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!
- Но использования типов данных
 - облегчает рассуждение о генерации входных данных
 - сводим к задаче генерации значений определённого класса зависимых типов

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!
- Но использования типов данных
 - облегчает рассуждение о генерации входных данных
 - сводим к задаче генерации значений определённого класса зависимых типов
 - снимает с нас задачу проверки корректности

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!
- Но использования типов данных
 - облегчает рассуждение о генерации входных данных
 - сводим к задаче генерации значений определённого класса зависимых типов
 - снимает с нас задачу проверки корректности
 - не можем создать значение, нарушающее спецификацию, оно не скомпилируется

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!
- Но использования типов данных
 - облегчает рассуждение о генерации входных данных
 - сводим к задаче генерации значений определённого класса зависимых типов
 - снимает с нас задачу проверки корректности
 - не можем создать значение, нарушающее спецификацию, оно не скомпилируется
 - облегчает последующую обработку сгенерированных данных

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!
- Но использования типов данных
 - облегчает рассуждение о генерации входных данных
 - сводим к задаче генерации значений определённого класса зависимых типов
 - снимает с нас задачу проверки корректности
 - не можем создать значение, нарушающее спецификацию, оно не скомпилируется
 - облегчает последующую обработку сгенерированных данных
 - и сами данные, и ограничения лежат рядом

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!
- Но использования типов данных
 - облегчает рассуждение о генерации входных данных
 - сводим к задаче генерации значений определённого класса зависимых типов
 - снимает с нас задачу проверки корректности
 - не можем создать значение, нарушающее спецификацию, оно не скомпилируется
 - облегчает последующую обработку сгенерированных данных
 - и сами данные, и ограничения лежат рядом
 - при обработке данных мы имеем все гарантии из ограничений

Может возникнуть вопрос

- Зачем зависимые типы?
- Существуют ли другие формализмы, способные описать то же самое?
- Конечно!
- Но использования типов данных
 - облегчает рассуждение о генерации входных данных
 - сводим к задаче генерации значений определённого класса зависимых типов
 - снимает с нас задачу проверки корректности
 - не можем создать значение, нарушающее спецификацию, оно не скомпилируется
 - облегчает последующую обработку сгенерированных данных
 - и сами данные, и ограничения лежат рядом
 - при обработке данных мы имеем все гарантии из ограничений
 - не надо обрабатывать несуществующие случаи

Сперва ○○	Property-based ○○○○○	Type-driven ○○○○○	Functional ○○○	Dependent ○○○○	Всё вместе ○○	Опыт, ○○○○○○	опыт, ○○	и ещё раз опыт ○○○○○	Напоследок ○●○
--------------	-------------------------	----------------------	-------------------	-------------------	------------------	-----------------	-------------	-------------------------	-------------------

- Property-based testing

Сперва
○○

Property-based
○○○○○

Type-driven
○○○○○

Functional
○○○

Dependent
○○○○

Всё вместе
○○

Опыт,
○○○○○○

опыт,
○○

и ещё раз опыт
○○○○○

Напоследок
○●○

- Property-based testing

✓ *хороший* метод

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения

- Property-based testing

- ✓ *хороший* метод

- ✗ требует освоения

- ✗ требует серьёзного взгляда на требования и формализацию

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ прекрасны

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны
- ✗ высокий уровень входа

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны
- ✗ высокий уровень входа
- ✗ нет в мейнстриме (пока?)

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны
- ✗ высокий уровень входа
- ✗ нет в мейнстриме (пока?)
- ✓ полезны не только в качестве спецификации

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны
- ✗ высокий уровень входа
- ✗ нет в мейнстриме (пока?)
- ✓ полезны не только в качестве спецификации

- Они вместе, как это ни удивительно, работают

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны
- ✗ высокий уровень входа
- ✗ нет в мейнстриме (пока?)
- ✓ полезны не только в качестве спецификации

- Они вместе, как это ни удивительно, работают

- ✓ позволяют тестировать то, что тяжело

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны
- ✗ высокий уровень входа
- ✗ нет в мейнстриме (пока?)
- ✓ полезны не только в качестве спецификации

- Они вместе, как это ни удивительно, работают

- ✓ позволяют тестировать то, что тяжело
- ✗ инструментальная поддержка на зачаточном уровне

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны
- ✗ высокий уровень входа
- ✗ нет в мейнстриме (пока?)
- ✓ полезны не только в качестве спецификации

- Они вместе, как это ни удивительно, работают

- ✓ позволяют тестировать то, что тяжело
- ✗ инструментальная поддержка на зачаточном уровне
- ✗ методы спецификации ещё не отработаны

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны
- ✗ высокий уровень входа
- ✗ нет в мейнстриме (пока?)
- ✓ полезны не только в качестве спецификации

- Они вместе, как это ни удивительно, работают

- ✓ позволяют тестировать то, что тяжело
- ✗ инструментальная поддержка на зачаточном уровне
- ✗ методы спецификации ещё не отработаны
- ✗ есть проблемы со скоростью работы

- Property-based testing

- ✓ *хороший* метод
- ✗ требует освоения
- ✗ требует серьёзного взгляда на требования и формализацию
- ✓ экономически оправдан для сложных и ответственных систем

- Зависимые типы

- ✓ мощны и выразительны
- ✗ высокий уровень входа
- ✗ нет в мейнстриме (пока?)
- ✓ полезны не только в качестве спецификации

- Они вместе, как это ни удивительно, работают

- ✓ позволяют тестировать то, что тяжело
- ✗ инструментальная поддержка на зачаточном уровне
- ✗ методы спецификации ещё не отработаны
- ✗ есть проблемы со скоростью работы
- ✓ мы только в начале пути

Если стало интересно



DepTyCheck, примеры



Эта презентация

Спасибо!



DepTyCheck, примеры



Эта презентация

Вопросы?