

# Генерация сложных тестовых данных со сложными инвариантами

Проблемы и *направления* решений

Денис Буздалов

24.11.2020

# Раздел 1

## Вступление

# Дисклеймеры

- Проблемы и *лишь направления* решений

# Дисклеймеры

- Проблемы и *лишь направления* решений
- Да и проблемы-то не все

# Дисклеймеры

- Проблемы и *лишь направления* решений
- Да и проблемы-то не все
  
- Ещё

# Дисклеймеры

- Проблемы и *лишь направления* решений
- Да и проблемы-то не все
  
- Ещё
- я

# Дисклеймеры

- Проблемы и *лишь направления* решений
- Да и проблемы-то не все
  
- Ещё
- я
- очень

# Дисклеймеры

- Проблемы и *лишь направления* решений
- Да и проблемы-то не все
  
- Ещё
- я
- очень
- люблю

# Дисклеймеры

- Проблемы и *лишь направления* решений
- Да и проблемы-то не все
  
- Ещё
- я
- очень
- люблю
- постепенно

# Дисклеймеры

- Проблемы и *лишь направления* решений
- Да и проблемы-то не все
  
- Ещё
- я
- очень
- люблю
- постепенно
- вываливающиеся

# Дисклеймеры

- Проблемы и *лишь направления* решений
- Да и проблемы-то не все
  
- Ещё
- я
- очень
- люблю
- постепенно
- вываливающиеся
- СПИСКИ

# Структура доклада

- Проблемы и направления

# Структура доклада

- Проблемы и направления
  - ▶ Не требует специальных знаний
  - ▶ Но конкретики мало

# Структура доклада

- Проблемы и направления
  - ▶ Не требует специальных знаний
  - ▶ Но конкретики мало
- Демонстрация многим незнакомого направления

# Структура доклада

- Проблемы и направления
  - ▶ Не требует специальных знаний
  - ▶ Но конкретики мало
- Демонстрация многим незнакомого направления
  - ▶ Конкретики будет больше, чем может хотеться

# Структура доклада

- Проблемы и направления
  - ▶ Не требует специальных знаний
  - ▶ Но конкретики мало
- Демонстрация многим незнакомого направления
  - ▶ Конкретики будет больше, чем может хотеться
  - ▶ Потребуется чтения нетривиального кода на нетривиальном и малознакомом языке программирования.
  - ▶ Знакомство в чистом функциональном программировании сильно поможет

## Раздел 2

### Проблемы и задачи

# Контекст задачи

# Контекст задачи тестирования

# Контекст задачи тестирования

- Система со сложными входными данными

# Контекст задачи тестирования

- Система со сложными входными данными

*напр., компилятор*

# Контекст задачи тестирования

- Система со сложными входными данными
- Корректность входных данных

*напр., компилятор*

# Контекст задачи тестирования

- Система со сложными входными данными *напр., компилятор*
- Корректность входных данных *напр., типизированность или объявленность переменных*

# Контекст задачи тестирования

- Система со сложными входными данными *напр., компилятор*
- Корректность входных данных *напр., типизированность или объявленность переменных*
- Этого свойство может быть непростым

# Контекст задачи тестирования

- Система со сложными входными данными *напр., компилятор*
- Корректность входных данных *напр., типизированность или объявленность переменных*
- Этого свойство может быть непростым
- Разные режимы работы системы на корректных входах

# Контекст задачи тестирования

- Система со сложными входными данными *напр., компилятор*
- Корректность входных данных *напр., типизированность или объявленность переменных*
- Этого свойство может быть непростым
- Разные режимы работы системы на корректных входах *напр., оптимизации компилятора*

# Контекст задачи тестирования

- Система со сложными входными данными *напр., компилятор*
- Корректность входных данных *напр., типизированность или объявленность переменных*
- Этого свойство может быть непростым
- Разные режимы работы системы на корректных входах *напр., оптимизации компилятора*
- Представляем дополнительные ограничения для режимов

# Контекст задачи тестирования

- Система со сложными входными данными *напр., компилятор*
- Корректность входных данных *напр., типизированность или объявленность переменных*
- Этого свойство может быть непростым
- Разные режимы работы системы на корректных входах *напр., оптимизации компилятора*
- Представляем дополнительные ограничения для режимов *напр., определённые виды оптимизаций компилятора*

# Контекст задачи тестирования

- Система со сложными входными данными *напр., компилятор*
- Корректность входных данных *напр., типизированность или объявленность переменных*
- Этого свойство может быть непростым
- Разные режимы работы системы на корректных входах *напр., оптимизации компилятора*
- Представляем дополнительные ограничения для режимов *напр., определённые виды оптимизаций компилятора*
- Хотим автоматизировать тестирование режимов работы системы на корректных входах

# Контекст задачи тестирования

- Система со сложными входными данными *напр., компилятор*
- Корректность входных данных *напр., типизированность или объявленность переменных*
- Этого свойство может быть непростым
- Разные режимы работы системы на корректных входах *напр., оптимизации компилятора*
- Представляем дополнительные ограничения для режимов *напр., определённые виды оптимизаций компилятора*
- Хотим автоматизировать тестирование режимов работы системы на корректных входах
- Задача генерации сложных тестовых данных со сложными ограничениями (инвариантами)

# Проблемы

# Проблемы

- Откуда взять сложные по структуре данные?

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант
  - ▶ консистентность изменений

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант
  - ▶ консистентность изменений
- Количество нетривиальных случаев

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант
  - ▶ консистентность изменений
- Количество нетривиальных случаев
  - ▶ случайная генерация: хорошее распределение

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант
  - ▶ консистентность изменений
- Количество нетривиальных случаев
  - ▶ случайная генерация: хорошее распределение
  - ▶ полная генерация: хорошее упорядочивание

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант
  - ▶ консистентность изменений
- Количество нетривиальных случаев
  - ▶ случайная генерация: хорошее распределение
  - ▶ полная генерация: хорошее упорядочивание
- Сложная структура самих инвариантов

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант
  - ▶ консистентность изменений
- Количество нетривиальных случаев
  - ▶ случайная генерация: хорошее распределение
  - ▶ полная генерация: хорошее упорядочивание
- Сложная структура самих инвариантов
  - ▶ например, предикат первого порядка (с кванторами)

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант
  - ▶ консистентность изменений
- Количество нетривиальных случаев
  - ▶ случайная генерация: хорошее распределение
  - ▶ полная генерация: хорошее упорядочивание
- Сложная структура самих инвариантов
  - ▶ например, предикат первого порядка (с кванторами)
  - ▶ например, произвольная функция на языке программирования

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант
  - ▶ консистентность изменений
- Количество нетривиальных случаев
  - ▶ случайная генерация: хорошее распределение
  - ▶ полная генерация: хорошее упорядочивание
- Сложная структура самих инвариантов
  - ▶ например, предикат первого порядка (с кванторами)
  - ▶ например, произвольная функция на языке программирования
- Эффективность генерации

# Проблемы

- Откуда взять сложные по структуре данные?
  - ▶ мы уже напрягались описывая сложную структуру и инвариант
  - ▶ консистентность изменений
- Количество нетривиальных случаев
  - ▶ случайная генерация: хорошее распределение
  - ▶ полная генерация: хорошее упорядочивание
- Сложная структура самих инвариантов
  - ▶ например, предикат первого порядка (с кванторами)
  - ▶ например, произвольная функция на языке программирования
- Эффективность генерации
- Всяк сверчок знает только свой шесток

## Раздел 3

### К решениям

# Направления решений

- Выделение полезных подклассов спецификаций
- Метрики сложности структуры данных
- Ленивость + немного магии
- Избавиться от инвариантов
- Конечно, многое другое...

# Полезные подклассы: примеры для языков

- Правила вывода

$$(VAR) \frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma} \quad (LAM) \frac{x:\sigma, \Gamma \vdash M:\tau}{\Gamma \vdash \lambda x:\sigma.M:\sigma \rightarrow \tau}$$

$$(APP) \frac{\Gamma \vdash M:\sigma \rightarrow \tau \quad \Gamma \vdash N:\sigma}{\Gamma \vdash MN:\tau}$$

- Описание систем типов а-ля *PLT Redex*
- Описание семантики а-ля *K framework*

# Сложность данных

- Нумерация по количеству конструкторов ADT

```
data Tree a = L | N a (Tree a) (Tree a)
```

```
N 4 (N 2 (N 1 (N 0 L L) (N 3 L L)) (N 6 (N 5 L L) (N 7 L L)))  
N 0 L (N 1 L (N 2 L (N 3 L (N 4 L (N 5 L (N 6 L (N 7 L L))))))))
```

# Сложность данных

- Нумерация по количеству конструкторов ADT

```
type Name = String
```

```
data Program = New Name Program | Name := Expr | Skip  
| Program :>> Program | If Expr Program Program  
| While Expr Program
```

```
data Expr = Var Name | Add Expr Expr
```

# Сложность данных

- Нумерация по количеству конструкторов ADT

```
type Name = String
```

```
data Program = New Name Program | Name := Expr | Skip  
| Program :>> Program | If Expr Program Program  
| While Expr Program
```

```
data Expr = Var Name | Add Expr Expr
```

- Возможность автоматически получать равномерное распределение среди примерно одинаково сложных экземпляров данных
- Позволяет подстраивать, например, назначая веса конструкторам

# Ленивость + магия

- Раннее отсечение вариантов

# Ленивость + магия

- Раннее отсечение вариантов
- Немного магии

```
valid :: (a → Bool) → Maybe Bool
```

# Ленивость + магия

- Раннее отсечение вариантов
- Немного магии

```
valid :: (a → Bool) → Maybe Bool
```

```
inspectsRight :: ((a, b) → Bool) → Bool
```

```
index p (a :: b) k i = if inspectsRight p  
  then index p (a *** b)          k i  
  else index p (swap :$: (b *** a)) k i
```

# Ленивость + магия

- Раннее отсечение вариантов
- Немного магии

```
valid :: (a → Bool) → Maybe Bool
```

```
inspectsRight :: ((a, b) → Bool) → Bool
```

```
index p (a :: b) k i = if inspectsRight p  
  then index p (a *** b)          k i  
  else index p (swap :$: (b *** a)) k i
```

- Существенно повышает эффективность при использовании произвольных функций в качестве предикатов

# Без инвариантов

- Сложные системы типов
- Может быть непросто с непривычки
- Спецификация всего в одном месте
- Консистентность объявлений
- Соответствие Карри-Ховарда

## Раздел 4

### Демонстрация

# Пример: примитивный императивный язычок

Волюнтаристские решения для иллюстрации:

# Пример: примитивный императивный язычок

Волюнтаристские решения для иллюстрации:

- Внешняя система типов данных

# Пример: примитивный императивный язычок

Волюнтаристские решения для иллюстрации:

- Внешняя система типов данных
- Ничего лишнего ;-)
  - ▶ Без неявного приведения типов
  - ▶ Без динамической памяти
  - ▶ Без функций и рекурсии

# Пример: примитивный императивный язычок

Волюнтаристские решения для иллюстрации:

- Внешняя система типов данных
- Ничего лишнего ;-)
  - ▶ Без неявного приведения типов
  - ▶ Без динамической памяти
  - ▶ Без функций и рекурсии
- Базовые конструкции: присваивание, ветвление, цикл

# Пример: примитивный императивный язычок

Волюнтаристские решения для иллюстрации:

- Внешняя система типов данных
- Ничего лишнего ;-)
  - ▶ Без неявного приведения типов
  - ▶ Без динамической памяти
  - ▶ Без функций и рекурсии
- Базовые конструкции: присваивание, ветвление, цикл
- Чёткое разделение на выражения и statement'ы

# Пример: примитивный императивный язычок

Волюнтаристские решения для иллюстрации:

- Внешняя система типов данных
- Ничего лишнего ;-)
  - ▶ Без неявного приведения типов
  - ▶ Без динамической памяти
  - ▶ Без функций и рекурсии
- Базовые конструкции: присваивание, ветвление, цикл
- Чёткое разделение на выражения и statement'ы
- Интересующие нас инварианты

# Пример: примитивный императивный язычок

Волюнтаристские решения для иллюстрации:

- Внешняя система типов данных
- Ничего лишнего ;-)
  - ▶ Без неявного приведения типов
  - ▶ Без динамической памяти
  - ▶ Без функций и рекурсии
- Базовые конструкции: присваивание, ветвление, цикл
- Чёткое разделение на выражения и statement'ы
- Интересующие нас инварианты
  - ▶ использование только объявленных переменных

# Пример: примитивный императивный язычок

Волюнтаристские решения для иллюстрации:

- Внешняя система типов данных
- Ничего лишнего ;-)
  - ▶ Без неявного приведения типов
  - ▶ Без динамической памяти
  - ▶ Без функций и рекурсии
- Базовые конструкции: присваивание, ветвление, цикл
- Чёткое разделение на выражения и statement'ы
- Интересующие нас инварианты
  - ▶ использование только объявленных переменных
  - ▶ использование переменных в соответствии с объявленным типом

# Пример: примитивный императивный язычок

Волюнтаристские решения для иллюстрации:

- Внешняя система типов данных
- Ничего лишнего ;-)
  - ▶ Без неявного приведения типов
  - ▶ Без динамической памяти
  - ▶ Без функций и рекурсии
- Базовые конструкции: присваивание, ветвление, цикл
- Чёткое разделение на выражения и statement'ы
- Интересующие нас инварианты
  - ▶ использование только объявленных переменных
  - ▶ использование переменных в соответствии с объявленным типом
- Например, **не** рассматриваем:
  - ▶ использование только определённых переменных (со значением)
  - ▶ отсутствие деления на ноль для целых чисел

# Примитивный императивный язычок

Интересующие нас инварианты

- использование только объявленных переменных
- использование объявленных переменных в соответствии с объявленным типом

# Примитивный императивный язычок

Интересующие нас инварианты

- использование только объявленных переменных
- использование объявленных переменных в соответствии с объявленным типом

```
data Name = MkName String
```

```
Context : Type
```

```
Context = List (Name, Type)
```

# Примитивный императивный язычок

**data** Statement : (pre : Context) → (post : Context) → Type **where**

# Примитивный императивный язычок

```
data Statement : (pre : Context) → (post : Context) → Type where  
  nop  : Statement ctx ctx
```

# Примитивный императивный язычок

```
data Statement : (pre : Context) → (post : Context) → Type where  
  nop   : Statement ctx ctx  
  (.)   : (0 ty : Type) → (n : Name) → Statement ctx $ (n, ty)::ctx
```

# Примитивный императивный язык

```
data Statement : (pre : Context) → (post : Context) → Type where
  nop      : Statement ctx ctx
  (.)      : (0 ty : Type) → (n : Name) → Statement ctx $ (n, ty)::ctx
  (#=)     : (n : Name) → (0 ty : Lookup n ctx) ⇒
             (v : Expression ctx $ reveal ty) → Statement ctx ctx
```

# Примитивный императивный язычок

```
data Statement : (pre : Context) → (post : Context) → Type where
  nop   : Statement ctx ctx
  (.)   : (0 ty : Type) → (n : Name) → Statement ctx $ (n, ty)::ctx
  (#=)  : (n : Name) → (0 ty : Lookup n ctx) ⇒
          (v : Expression ctx $ reveal ty) → Statement ctx ctx
  for   : (init : Statement outer_ctx inside_for) →
          (cond : Expression inside_for Bool) →
          (upd  : Statement inside_for inside_for) →
          (body : Statement inside_for after_body) →
          Statement outer_ctx outer_ctx
```

# Примитивный императивный язычок

```
data Statement : (pre : Context) → (post : Context) → Type where
  nop  : Statement ctx ctx
  (.)  : (0 ty : Type) → (n : Name) → Statement ctx $ (n, ty)::ctx
  (#=) : (n : Name) → (0 ty : Lookup n ctx) ⇒
        (v : Expression ctx $ reveal ty) → Statement ctx ctx
  for  : (init : Statement outer_ctx inside_for) →
        (cond : Expression inside_for Bool) →
        (upd  : Statement inside_for inside_for) →
        (body : Statement inside_for after_body) →
        Statement outer_ctx outer_ctx
  if__ : (cond : Expression ctx Bool) →
        Statement ctx ctx_then → Statement ctx ctx_else →
        Statement ctx ctx
```

# Примитивный императивный язычок

```
data Statement : (pre : Context) → (post : Context) → Type where
  nop   : Statement ctx ctx
  (.)   : (0 ty : Type) → (n : Name) → Statement ctx $ (n, ty)::ctx
  (#=)  : (n : Name) → (0 ty : Lookup n ctx) ⇒
          (v : Expression ctx $ reveal ty) → Statement ctx ctx
  for   : (init : Statement outer_ctx inside_for) →
          (cond : Expression inside_for Bool) →
          (upd  : Statement inside_for inside_for) →
          (body : Statement inside_for after_body) →
          Statement outer_ctx outer_ctx
  if__  : (cond : Expression ctx Bool) →
          Statement ctx ctx_then → Statement ctx ctx_else →
          Statement ctx ctx
  (*>) : Statement pre mi → Statement mi post → Statement pre post
```

# Примитивный императивный язык

```
data Statement : (pre : Context) → (post : Context) → Type where
  nop  : Statement ctx ctx
  (.)  : (0 ty : Type) → (n : Name) → Statement ctx $ (n, ty)::ctx
  (#=) : (n : Name) → (0 ty : Lookup n ctx) ⇒
        (v : Expression ctx $ reveal ty) → Statement ctx ctx
  for  : (init : Statement outer_ctx inside_for) →
        (cond : Expression inside_for Bool) →
        (upd  : Statement inside_for inside_for) →
        (body : Statement inside_for after_body) →
        Statement outer_ctx outer_ctx
  if__ : (cond : Expression ctx Bool) →
        Statement ctx ctx_then → Statement ctx ctx_else →
        Statement ctx ctx
  (*>) : Statement pre mi → Statement mi post → Statement pre post
  block : Statement outer inside → Statement outer outer
```

# Примитивный императивный язык

```
data Statement : (pre : Context) → (post : Context) → Type where
  nop   : Statement ctx ctx
  (.)   : (0 ty : Type) → (n : Name) → Statement ctx $ (n, ty)::ctx
  (#=)  : (n : Name) → (0 ty : Lookup n ctx) ⇒
          (v : Expression ctx $ reveal ty) → Statement ctx ctx
  for   : (init : Statement outer_ctx inside_for) →
          (cond : Expression inside_for Bool) →
          (upd  : Statement inside_for inside_for) →
          (body : Statement inside_for after_body) →
          Statement outer_ctx outer_ctx
  if__  : (cond : Expression ctx Bool) →
          Statement ctx ctx_then → Statement ctx ctx_else →
          Statement ctx ctx
  (*>) : Statement pre mi → Statement mi post → Statement pre post
  block : Statement outer inside → Statement outer outer
  print : Show ty ⇒ Expression ctx ty → Statement ctx ctx
```

# Примитивный императивный язычок

```
data Expression : (ctx : Context) → (res : Type) → Type where
```

# Примитивный императивный язычок

```
data Expression : (ctx : Context) → (res : Type) → Type where  
  -- Constant expression  
  C : (x : ty) → Expression ctx ty
```

# Примитивный императивный язычок

```
data Expression : (ctx : Context) → (res : Type) → Type where
  -- Constant expression
  C : (x : ty) → Expression ctx ty
  -- Value of the variable
  V : (n : Name) → (0 ty : Lookup n ctx) ⇒
    Expression ctx $ reveal ty
```

# Примитивный императивный язычок

```
data Expression : (ctx : Context) → (res : Type) → Type where
  -- Constant expression
  C : (x : ty) → Expression ctx ty
  -- Value of the variable
  V : (n : Name) → (0 ty : Lookup n ctx) ⇒
    Expression ctx $ reveal ty
  -- Unary operation over the result of an expression
  U : (f : a → b) → Expression ctx a → Expression ctx b
```

# Примитивный императивный язычок

```
data Expression : (ctx : Context) → (res : Type) → Type where
  -- Constant expression
  C : (x : ty) → Expression ctx ty
  -- Value of the variable
  V : (n : Name) → (0 ty : Lookup n ctx) ⇒
      Expression ctx $ reveal ty
  -- Unary operation over the result of an expression
  U : (f : a → b) → Expression ctx a → Expression ctx b
  -- Binary operation over the results of two expressions
  B : (f : a → b → c) → Expression ctx a → Expression ctx b →
      Expression ctx c
```

# Примеры кода

```
simple_ass : Statement ctx $ ("x", Int)::ctx  
simple_ass = do  
  Int. "x"  
  "x" #= C 2
```

## Примеры кода

Для честности, нужно упомянуть лифтинг функций

`(+)` : `Expression ctx Int → Expression ctx Int → Expression ctx Int`

`(+) = B (+)`

`(<)` : `Expression ctx Int → Expression ctx Int → Expression ctx Bool`

`(<) = B (<)`

`(&&)` : `Expression ctx Bool → Expression ctx Bool → Expression ctx Bool`

`(&&) = B (\a, b ⇒ a && b) -- recoded because of laziness`

`(≠)` : `Eq a ⇒ Expression ctx a → Expression ctx a →`

`Expression ctx Bool`

`(≠) = B (≠)`

...

## Примеры кода

```
lost_block : Statement ctx ctx
lost_block = do
  block $ do
    Int. "x"
    "x" #= C 2
    Int. "y" #= V "x"
    Int. "z" #= C 3
    print $ V "y" + V "z" + V "x"
```

## Примеры кода

```
lost_block : Statement ctx ctx
lost_block = do
  block $ do
    Int. "x"
    "x" #= C 2
    Int. "y" #= V "x"
    Int. "z" #= C 3
    print $ V "y" + V "z" + V "x"
print $ V "y" + C 1
```

## Примеры кода

```
lost_block : Statement ctx ctx
lost_block = do
  block $ do
    Int. "x"
    "x" #= C 2
    Int. "y" #= V "x"
    Int. "z" #= C 3
    print $ V "y" + V "z" + V "x"
  print $ V "y" + C 1
```

Error: While processing right hand side of lost\_block.  
Can't find an implementation for Lookup (MkName "y") ctx.

```
164 | print $ V "y" + C 1
      |         ^^^^
```

## Примеры кода

```
name_shadowing : Statement ctx ctx
name_shadowing = block $ do
  Int. "x" #= C 0
  block $ do
    Int. "x" #= C 3
    Int. "y" #= V "x" + C 2
    String. "x" #= C "foo"
    print $ V "x" ++ C "bar" ++ show (V "y")
  Int. "z" #= V "x" + C 2
```

## Примеры кода

```
some_for : Statement ctx ctx
some_for = for (do Int. "x" != C 0; Int. "y" != C 0)
              (V "x" < C 5 && V "y" < C 10)
              ("x" != V "x" + C 1) $ do
                "y" != V "y" + V "x" + C 1
```

## Примеры кода

```
some_for : Statement ctx ctx
some_for = for (do Int. "x" != C 0; Int. "y" != C 0)
  (V "x" < C 5 && V "y" < C 10)
  ("x" != V "x" + C 1) $ do
    "y" != V "y" + V "x" + C 1
```

```
bad_for : Statement ctx ctx
bad_for = for (do Int. "x" != C 0; Int. "y" != C 0)
  (V "y")
  ("x" != V "x" + C 1) $ do
    "y" != V "y" `div` V "x" + C 1
```

## Примеры кода

```
some_for : Statement ctx ctx
some_for = for (do Int. "x" != C 0; Int. "y" != C 0)
              (V "x" < C 5 && V "y" < C 10)
              ("x" != V "x" + C 1) $ do
              "y" != V "y" + V "x" + C 1
```

```
bad_for : Statement ctx ctx
bad_for = for (do Int. "x" != C 0; Int. "y" != C 0)
              (V "y")
              ("x" != V "x" + C 1) $ do
              "y" != V "y" `div` V "x" + C 1
```

Error: While processing right hand side of bad\_for.  
Mismatch between: Int and Bool.

171

```
(V "y")
^^^^^
```

## Примеры кода

```
while : Expression ctx Bool → Statement ctx after_body →  
      Statement ctx ctx  
while cond = for nop cond nop
```

```
euc : {0 ctx : Context} → let c = ("a", Int)::("b", Int)::ctx in  
    Statement c $ ("res", Int)::c  
euc = do  
  while (V "a" ≠ C 0 && V "b" ≠ C 0) $ do  
    if__ (V "a" > V "b")  
      ("a" ≠ V "a" `mod` V "b")  
      ("b" ≠ V "b" `mod` V "a")  
  Int. "res" ≠ V "a" + V "b"
```

## Волшебное свойство, на самом деле, простое

$(\#=) : (n : \text{Name}) \rightarrow (\emptyset \text{ ty} : \text{Lookup } n \text{ ctx}) \Rightarrow$   
 $(v : \text{Expression } \text{ctx } \$ \text{ reveal } \text{ty}) \rightarrow \text{Statement } \text{ctx } \text{ctx}$

$V : (n : \text{Name}) \rightarrow (\emptyset \text{ ty} : \text{Lookup } n \text{ ctx}) \Rightarrow$   
 $\text{Expression } \text{ctx } \$ \text{ reveal } \text{ty}$

## Волшебное свойство, на самом деле, простое

```
(#=) : (n : Name) → (0 ty : Lookup n ctx) ⇒  
      (v : Expression ctx $ reveal ty) → Statement ctx ctx
```

```
V : (n : Name) → (0 ty : Lookup n ctx) ⇒  
    Expression ctx $ reveal ty
```

---

```
data Lookup : a → List (a, b) → Type where  
  There : Lookup z xys → Lookup z $ (x, y)::xys  
  Here  : (y : b) → Lookup x $ (x, y)::xys
```

```
reveal : Lookup {b} x xys → b  
reveal (Here y) = y  
reveal (There subl) = reveal subl
```

# Рукописный генератор

```
varExprGen' : {a : Type} → {ctx : Context} → DecEq' Type ⇒ List (Expression ctx a)
varExprGen' = map varExpr varsOfType where
  varExpr : (n : Name ** lk : Lookup n ctx ** reveal lk = a) → Expression ctx a
  varExpr (n ** _ ** prf) = rewrite sym prf in V n

varsOfType : List (n : Name ** lk : Lookup n ctx ** reveal lk = a)
varsOfType = varsOfTypeOfCtx $ addLookups ctx
  where
    addLookups : (ctx : Context) → List (n : Name ** ty : Type ** lk : Lookup n ctx ** reveal lk = ty)
    addLookups [] = []
    addLookups ((n, ty)::xs) = (n ** ty ** Here ty ** Refl) ::
      map (\(n ** ty ** lk ** lk_ty) ⇒ (n ** ty ** There lk ** lk_ty)) (addLookups xs)

varsOfTypeOfCtx : List (n : Name ** ty : Type ** lk : Lookup n ctx ** reveal lk = ty) →
  List (n : Name ** lk : Lookup n ctx ** reveal lk = a)
varsOfTypeOfCtx [] = []
varsOfTypeOfCtx ((n ** ty ** lk ** lk_ty)::xs) = maybeToList varX ++ varsOfTypeOfCtx xs where
  varX : Maybe (n : Name ** lk : Lookup n ctx ** reveal lk = a)
  varX = case decEq' ty a of
    (Yes ty_a) ⇒ Just (n ** lk ** trans lk_ty ty_a)
    No ⇒ Nothing

commonGens : {a : Type} → {ctx : Context} → Gen a → DecEq' Type ⇒ (n ** Vect n $ Gen $ Expression ctx a)
commonGens g = ( _ ** [C <$> g] ++ map pure (fromList varExprGen') )

exprGen : (szBound : Nat) → {a : Type} → Gen a → Gen (a → a) → Gen (a → a → a) → {ctx : Context} → DecEq' Type ⇒
  Gen (Expression ctx a)
exprGen Z g _ _ = oneOf $ snd $ commonGens g
exprGen (S n) g gg ggg = oneOf $ snd (commonGens g) ++
  [ [| U gg (exprGen n g gg ggg) |]
  , let s = exprGen n g gg ggg in [| B ggg s s |]
  ]
```

# Рукописный генератор

```
lookupGen : (ctx : Context) → NonEmpty ctx ⇒ Gen (n : Name ** Lookup n ctx)
lookupGen ctx = let (lks@(::_) ** _) = mapLk ctx in oneOf $ map pure $ fromList lks where
  mapLk : (ctx : Context) → NonEmpty ctx ⇒ (l : List (n : Name ** Lookup n ctx) ** NonEmpty l)
  mapLk [(n, ty)] = ( [(n ** Here ty)] ** IsNonEmpty )
  mapLk ((n, ty)::xs@(::_)) = ( (n ** Here ty) :: map (\(n ** lk) ⇒ (n ** There lk)) (fst $ mapLk xs) ** IsNonEmpty )
```

## mutual

```
noDeclStmtGen : (ctx : Context) → Gen Type ⇒ Gen Name ⇒
  (genExpr : {a : Type} → {ctx : Context} → Gen (Expression ctx a)) ⇒ Gen (Statement ctx ctx)
```

```
noDeclStmtGen ctx = oneOf
  [ pure nop
  , case ctx of
    [] ⇒ pure nop
    (::_) ⇒ do (n ** _) ← lookupGen ctx
               pure $ n #≠ !genExpr
  , do (inside_for ** init) ← stmtGen ctx
       (_ ** body) ← stmtGen inside_for
       pure $ for init !genExpr !(noDeclStmtGen inside_for) body
  , pure $ if__ !genExpr (snd !(stmtGen ctx)) (snd !(stmtGen ctx))
  , pure $ !(noDeclStmtGen ctx) *> !(noDeclStmtGen ctx)
  , pure $ block $ snd !(stmtGen ctx)
  , pure $ print !(genExpr {a=String}) ]
```

```
stmtGen : (pre : Context) → (genTy : Gen Type) ⇒ (genName : Gen Name) ⇒
  ({a : Type} → {ctx : Context} → Gen (Expression ctx a)) ⇒ Gen (post ** Statement pre post)
```

```
stmtGen pre = oneOf
  [ do s ← noDeclStmtGen pre
    pure (pre ** s)
  , do ty ← genTy
       n ← genName
       pure ((n, ty)::pre ** ty. n)
  , do (mid ** l) ← stmtGen pre
       (post ** r) ← stmtGen mid
       pure (post ** l *> r) ]
```

# Итог

- Описание декларативной семантики feasible

# Итог

- Описание декларативной семантики feasible
- В типах фактически закодированы правила вывода

# Итог

- Описание декларативной семантики feasible
- В типах фактически закодированы правила вывода
- Выражение инварианта “рядом”

# Итог

- Описание декларативной семантики *feasible*
- В типах фактически закодированы правила вывода
- Выражение инварианта “рядом”
- Код инвариантов может “поглотить” декларативную семантику (*вспомните Java-вставки в ANTLR и грамматику Алгол 68*)

# Итог

- Описание декларативной семантики *feasible*
- В типах фактически закодированы правила вывода
- Выражение инварианта “рядом”
- Код инвариантов может “поглотить” декларативную семантику (*вспомните Java-вставки в ANTLR и грамматику Алгол 68*)
- Рукописный генератор сложнее, чем описание

# Итог

- Описание декларативной семантики *feasible*
- В типах фактически закодированы правила вывода
- Выражение инварианта “рядом”
- Код инвариантов может “поглотить” декларативную семантику (*вспомните Java-вставки в ANTLR и грамматику Алгол 68*)
- Рукописный генератор сложнее, чем описание
- Нужна автоматизация генерации генераторов

# Итог

- Описание декларативной семантики *feasible*
- В типах фактически закодированы правила вывода
- Выражение инварианта “рядом”
- Код инвариантов может “поглотить” декларативную семантику (*вспомните Java-вставки в ANTLR и грамматику Алгол 68*)
- Рукописный генератор сложнее, чем описание
- Нужна автоматизация генерации генераторов
- После автоматизации получаем мощный и эффективный метод относительно задёшево

## Раздел 5

### Заключение

# Заключение

*Дороги трудны, но хуже без дорог  
Ю. Визбор*

# Заключение

*Дороги трудны, но хуже без дорог  
Ю. Визбор*

- Двигаться есть куда

# Заключение

*Дороги трудны, но хуже без дорог  
Ю. Визбор*

- Двигаться есть куда
- Лучшее, по-видимому, всегда в смешении идей
  - ▶ автоматизация генерации генераторов
  - ▶ хорошее распределение
  - ▶ сложные свойства в самих структурах данных
  - ▶ скорость работы

# Заключение

*Дороги трудны, но хуже без дорог  
Ю. Визбор*

- Двигаться есть куда
- Лучшее, по-видимому, всегда в смешении идей
  - ▶ автоматизация генерации генераторов
  - ▶ хорошее распределение
  - ▶ сложные свойства в самих структурах данных
  - ▶ скорость работы
- Зависимые типы восхитительны ;-)

# Заключение

*Дороги трудны, но хуже без дорог  
Ю. Визбор*

- Двигаться есть куда
- Лучшее, по-видимому, всегда в смешении идей
  - ▶ автоматизация генерации генераторов
  - ▶ хорошее распределение
  - ▶ сложные свойства в самих структурах данных
  - ▶ скорость работы
- Зависимые типы восхитительны ;- ) ...как минимум перспективны

- Michał Palka, Koen Claessen, Alejanrdo Russo, John Hughes. [Testing an Optimising Compiler by Generating Random Terms \(2011\)](#)
- Koen Claessen, Jonas Duregard, Michał Palka. [Generating Constrained Random Data with Uniform Distribution \(2014\)](#)
- Alexey Rodriguez Yakushev, Johan Jeuring. [Enumerating Well-Typed Terms Generically \(2009\)](#)
- Michał Palka. PhD thesis. [Random Structured Test Data Generation for Black-Box Testing \(2014\)](#)
- Ivory language team. [Guilt Free Ivory \(2015\)](#)
- Rohan Sharma, Milos Gligoric, Vilas Jagannath, Darko Marinov. [A Comparison of Constraint-based and Sequence-based Generation of Complex Input Data Structures \(2010\)](#)
- Darko Marinov. PhD thesis. [Automatic Testing of Software with Structurally Complex Inputs \(2004\)](#)
- Burke Fetscher, Koen Claessen, Michał Palka, John Hughes, Robert Bruce Findler. [Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System \(2015\)](#)
- Vsevolod Livinskii, Dmitry Babokin, John Regehr. [Random Testing for C and C++ Compilers with YARPGen \(2020\)](#)
- и многое-многое другое... *(элементы списка нажимабельны)*

Спасибо

Вопросы?