Функциональное программирование

Лекция 7

Степан Львович Кузнецов

НИУ ВШЭ, факультет компьютерных наук

Монады

- Напомним, что мы продолжаем обсуждать *монады.*

Монады

- Напомним, что мы продолжаем обсуждать *монады.*
- Монада в Haskell'e — это преобразование типов, превращающее произвольный тип `a` в тип `m a`.

Монады

- Напомним, что мы продолжаем обсуждать *монады.*
- Монада в Haskell'е — это преобразование типов, превращающее произвольный тип a в тип `m a`.
- Можно мыслить, что внутри объекта типа `m a` каким-то образом присутствуют объекты исходного типа a, и монадические операции позволяют некоторым образом работать с этими объектами.

Монады

- Во-первых, объект типа a можно преобразовать в объект типа `m a` — «положить объект в монаду»:

```
return :: Monad m => a -> m a
```

Монады

- Во-первых, объект типа a можно преобразовать в объект типа `m a` — «положить объект в монаду»:

  ```
  return :: Monad m => a -> m a
  ```

- Во-вторых, к объектам внутри монады можно применять функции, причём возможно с монадическим результатом (bind):

  ```
  (>>=) :: Monad m => m a -> (a -> m b) -> m b
  ```

Монады

- Во-первых, объект типа a можно преобразовать в объект типа `m a` — «положить объект в монаду»:

```
return :: Monad m => a -> m a
```

- Во-вторых, к объектам внутри монады можно применять функции, причём возможно с монадическим результатом (bind):

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

- В частности, можно применить «чистую» функцию:

```
fmap :: Functor m => (a -> b) -> m a -> m b
```

причём для монады `fmap f x = (x >>= return . f))`

Монады

- Во-первых, объект типа a можно преобразовать в объект типа `m a` — «положить объект в монаду»:

```
return :: Monad m => a -> m a
```

- Во-вторых, к объектам внутри монады можно применять функции, причём возможно с монадическим результатом (bind):

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

- В частности, можно применить «чистую» функцию:

```
fmap :: Functor m => (a -> b) -> m a -> m b
```

  причём для монады `fmap f x = (x >>= return . f))`
- А вот «извлечь объект из монады» в общем случае невозможно.

Примеры монад

- Монада `IO` для взаимодействия с «внешним миром».

Примеры монад

- Монада `IO` для взаимодействия с «внешним миром».
- Вычисление в монаде родственно императивному программированию, что ясно видно в do-нотации.

Примеры монад

- Монада `IO` для взаимодействия с «внешним миром».
- Вычисление в монаде родственно императивному программированию, что ясно видно в do-нотации.
- «Чистые» монады:

Примеры монад

- Монада `IO` для взаимодействия с «внешним миром».
- Вычисление в монаде родственно императивному программированию, что ясно видно в do-нотации.
- «Чистые» монады:
    - список,

Примеры монад

- Монада `IO` для взаимодействия с «внешним миром».
- Вычисление в монаде родственно императивному программированию, что ясно видно в do-нотации.
- «Чистые» монады:
    - список,
    - множество (для недетерминированных вычислений),

Примеры монад

- Монада `IO` для взаимодействия с «внешним миром».
- Вычисление в монаде родственно императивному программированию, что ясно видно в do-нотации.
- «Чистые» монады:
  - список,
  - множество (для недетерминированных вычислений),
  - вероятностные монады (монада Жири).

Монада Eval

- Ещё один пример монады связан с *порядком вычислений*.

Монада Eval

- Ещё один пример монады связан с *порядком вычислений.*
- Вспомним, что в Haskell'е есть операторы для изменения нормального порядка редукций.

Монада Eval

- Ещё один пример монады связан с *порядком вычислений.*
- Вспомним, что в Haskell'е есть операторы для изменения нормального порядка редукций.
- Таковые операторы — seq (строгость), par (параллелизм).

Монада Eval

- Более удобное средство работы с параллелизмом (и вообще со стратегиями редукций) даёт монада `Eval` из `Control.Parallel.Strategies`.

```
data Eval a = Done a
instance Monad Eval where
  return x = Done x
  Done x >>= k = k x   -- Note: pattern 'Done x' makes
                       -- '>>=' strict
```

Монада Eval

- Более удобное средство работы с параллелизмом (и вообще со стратегиями редукций) даёт монада `Eval` из `Control.Parallel.Strategies`.

```haskell
data Eval a = Done a
instance Monad Eval where
  return x = Done x
  Done x >>= k = k x   -- Note: pattern 'Done x' makes
                       -- '>>=' strict
```

  - Комментарий о строгости означает следующее: если первый аргумент `>>=` окажется `undefined`, то вычисление прервётся (он не отождествится с `Done x`).

Монада Eval

- Более удобное средство работы с параллелизмом (и вообще со стратегиями редукций) даёт монада `Eval` из `Control.Parallel.Strategies`.

```haskell
data Eval a = Done a
instance Monad Eval where
  return x = Done x
  Done x >>= k = k x    -- Note: pattern 'Done x' makes
                        -- '>>=' strict
```

  - Комментарий о строгости означает следующее: если первый аргумент `>>=` окажется `undefined`, то вычисление прервётся (он не отождествится с `Done x`).
- Смысл `Eval a` — объект типа `a`, вычисляемый с определённым порядком редукций.

Монада Eval

- В монаду `Eval` есть другие «входы», кроме стандартного `return`. Они имеют тип `Strategy a`, т.е. `a -> Eval a`

Монада Eval

- В монаду `Eval` есть другие «входы», кроме стандартного `return`. Они имеют тип `Strategy a`, т.е. `a -> Eval a`
- Например:

```
rpar x = x `par` return x
rseq x = x `pseq` return x
```

Монада Eval

- В монаду `Eval` есть другие «входы», кроме стандартного `return`. Они имеют тип `Strategy a`, т.е. `a -> Eval a`
- Например:

```
rpar x = x `par` return x
rseq x = x `pseq` return x
```

- В частности, `rpar` начинает вычисление `x` в параллельном потоке (spark’e), а также передаёт `x` для дальнейшего использования.

Монада Eval

- В монаду `Eval` есть другие «входы», кроме стандартного `return`. Они имеют тип `Strategy a`, т.е. `a -> Eval a`
- Например:

```
rpar x = x `par` return x
rseq x = x `pseq` return x
```

- В частности, `rpar` начинает вычисление `x` в параллельном потоке (spark'е), а также передаёт `x` для дальнейшего использования.
- При этом монада `Eval` является «чистой», и из неё есть «выход»:

```
runEval :: Eval a -> a
runEval (Done x) = x
```

Монада Eval

- Обычный `return`, он же `r0`, не вызывает вычисление аргумента, он сохраняется как thunk.

Монада Eval

- Обычный `return`, он же `r0`, не вызывает вычисление аргумента, он сохраняется как thunk.
- С помощью `Eval` можно реализовать параллельное применение `map` к элементам списка:

```haskell
parMap' :: (a -> b) -> [a] -> Eval [b]
parMap' f [] = return []
parMap' f (a:as) = do
    b <- rpar (f a)
    bs <- parMap' f as
    return (b:bs)
```

Пример: ParitySAT

```haskell
xor :: Bool -> Bool -> Bool
xor = (/=)

satPartial fm valset = foldr xor False $ map (fmEval fm) valset
partValSet partVals m = map (partVals ++) (allVals m)

n = 22
k = 4
m = n + 1 - k

xsat = runEval $ parMap' (satPartial (myFm n)) (map (partValSet m) (allVals k))

main = putStrLn $ show xsat
```

Свойства полиморфных функций

- Отвлечёмся от монад и посмотрим, какие свойства функции можно установить, опираясь только на её полиморфный тип, — так называемые свободные, или «бесплатные», теоремы *(free theorems).*

Свойства полиморфных функций

- Отвлечёмся от монад и посмотрим, какие свойства функции можно установить, опираясь только на её полиморфный тип, — так называемые свободные, или «бесплатные», теоремы *(free theorems).*
  - P. Wadler. Theorems for free! Proc. FPCA 1989

Свойства полиморфных функций

- Отвлечёмся от монад и посмотрим, какие свойства функции можно установить, опираясь только на её полиморфный тип, — так называемые свободные, или «бесплатные», теоремы *(free theorems).*
  - P. Wadler. Theorems for free! Proc. FPCA 1989
- Например, если $f : \forall r.(r \to r)$ и f *вычисляется без ошибки и за конечное время* на x, то, наверное, $f(x) = x$.

Свойства полиморфных функций

- Отвлечёмся от монад и посмотрим, какие свойства функции можно установить, опираясь только на её полиморфный тип, — так называемые свободные, или «бесплатные», теоремы *(free theorems).*
  - P. Wadler. Theorems for free! Proc. FPCA 1989
- Например, если $f : \forall r.(r \to r)$ и f *вычисляется без ошибки и за конечное время* на x, то, наверное, $f(x) = x$.
  - Действительно, тип f настолько общий, что она не может сделать с x'ом ничего содержательного, только вернуть его как было.

Свойства полиморфных функций

- Отвлечёмся от монад и посмотрим, какие свойства функции можно установить, опираясь только на её полиморфный тип, — так называемые свободные, или «бесплатные», теоремы *(free theorems).*
  - P. Wadler. Theorems for free! Proc. FPCA 1989
- Например, если $f : \forall r.(r \to r)$ и f *вычисляется без ошибки и за конечное время* на x, то, наверное, $f(x) = x$.
  - Действительно, тип f настолько общий, что она не может сделать с x'ом ничего содержательного, только вернуть его как было.
- Другой пример — $\mathbf{B} : \forall pqr.((q \to r) \to (p \to q) \to p \to r)$ может быть только операцией композиции — опять же, если не зависнет.

Свойства полиморфных функций

- Иногда по полиморфному типу невозможно точно определить поведение функции, но возможно установить какие-то её свойства.

Свойства полиморфных функций

- Иногда по полиморфному типу невозможно точно определить поведение функции, но возможно установить какие-то её свойства.
- Например, функция `f :: [a] -> [a]` может делать разные вещи со списком, но должна коммутировать с `map g`:
  `f . (map g) = (map g) . f`

Свойства полиморфных функций

- Иногда по полиморфному типу невозможно точно определить поведение функции, но возможно установить какие-то её свойства.
- Например, функция `f :: [a] -> [a]` может делать разные вещи со списком, но должна коммутировать с `map g`: `f . (map g) = (map g) . f`
- Действительно, для `f` элемены списка — это «чёрные ящики». Функция может их перетасовывать, но никак не может использовать то, что внутри.

Свойства полиморфных функций

- При этом другие желаемые свойства полиморфных функций вот так «за бесплатно» не получаются.

Свойства полиморфных функций

- При этом другие желаемые свойства полиморфных функций вот так «за бесплатно» не получаются.
- Например,

  ```
  map' :: (a -> b) -> [a] -> [b]
  ```

  не только не обязательно совпадает с «настоящим» `map`, но может и не удовлетворять законам функториальности.

Свойства полиморфных функций

- При этом другие желаемые свойства полиморфных функций вот так «за бесплатно» не получаются.
- Например,

  ```
  map' :: (a -> b) -> [a] -> [b]
  ```

  не только не обязательно совпадает с «настоящим» `map`, но может и не удовлетворять законам функториальности.
- Например,

  ```
  map' = \f -> ((map f) . reverse)
  ```

  нарушает оба закона: не переводит `id` в `id` и не коммутирует с композицией.

Свойства полиморфных функций

- При этом другие желаемые свойства полиморфных функций вот так «за бесплатно» не получаются.
- Например,

  ```
  map' :: (a -> b) -> [a] -> [b]
  ```

  не только не обязательно совпадает с «настоящим» `map`, но может и не удовлетворять законам функториальности.
- Например,

  ```
  map' = \f -> ((map f) . reverse)
  ```

  нарушает оба закона: не переводит `id` в `id` и не коммутирует с композицией.
- Как же выявлять и обосновывать истинные свободные теоремы?

Сопоставление элементов

- Пусть `f :: forall a. [a] -> [a]` и `g :: c -> d`, причём g биективна.

Сопоставление элементов

- Пусть `f :: forall a. [a] -> [a]` и `g :: c -> d`, причём `g` биективна.
- Заменим в типе `c` каждый элемент `x` на его образ `g x`.

Сопоставление элементов

- Пусть `f :: forall a. [a] -> [a]` и `g :: c -> d`, причём `g` биективна.
- Заменим в типе `c` каждый элемент `x` на его образ `g x`.
- Поскольку `f` определена *единообразно* для всех типов, подставляемых вместо `a`, она должна действовать после замены так же, как и до.

Сопоставление элементов

- Пусть `f :: forall a. [a] -> [a]` и `g :: c -> d`, причём `g` биективна.
- Заменим в типе `c` каждый элемент `x` на его образ `g x`.
- Поскольку `f` определена *единообразно* для всех типов, подставляемых вместо `a`, она должна действовать после замены так же, как и до.
- Это и означает, что `f . (map g) = (map g) . f`

Сопоставление элементов

- Пусть `f :: forall a. [a] -> [a]` и `g :: c -> d`, причём `g` биективна.
- Заменим в типе `c` каждый элемент `x` на его образ `g x`.
- Поскольку `f` определена *единообразно* для всех типов, подставляемых вместо `a`, она должна действовать после замены так же, как и до.
- Это и означает, что `f . (map g) = (map g) . f`
- Коммутативная диаграмма:

$$\begin{tikzcd}
[C] \arrow[r, "{[g]}"] \arrow[d, "f_C"'] & [D] \arrow[d, "f_D"] \\
[C] \arrow[r, "{[g]}"] & [D]
\end{tikzcd}$$

Отношения вместо функций

- В общем случае, если g не биективна, а f имеет более сложный тип, такой простой подход с «заменой типа» не работает.

Отношения вместо функций

- В общем случае, если g не биективна, а f имеет более сложный тип, такой простой подход с «заменой типа» не работает.
- Например, в тип композиции $(q \to r) \to (p \to q) \to p \to r$ переменная p входит как позитивно, так и негативно, и непонятно, в какую сторону рисовать стрелки:

$$
\begin{tikzcd}
p_2 \\
p_1 \arrow[u, "g"]
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
p_2 \to A \arrow[d, "A \to g"'] \\
p_1 \to A
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
(p_2 \to C) \to D \\
(p_1 \to C) \to D \arrow[u, "(g \to C) \to D"]
\end{tikzcd}
$$

Отношения вместо функций

- В общем случае, если g не биективна, а f имеет более сложный тип, такой простой подход с «заменой типа» не работает.
- Например, в тип композиции $(q \to r) \to (p \to q) \to p \to r$ переменная p входит как позитивно, так и негативно, и непонятно, в какую сторону рисовать стрелки:

$$
\begin{tikzcd}
p_2 & p_2 \to A \arrow[d, "A\to g"'] & (p_2 \to C) \to D \\
p_1 \arrow[u, "g"] & p_1 \to A & (p_1 \to C) \to D \arrow[u, "(g\to C)\to D"]
\end{tikzcd}
$$

- В процессе построения λ-терма могут возникать очень сложные типы.

Отношения вместо функций

- Вместо функций («стрелок») будем рассматривать *отношения* между типами, подставляемыми вместо данной переменной.

Отношения вместо функций

- Вместо функций («стрелок») будем рассматривать *отношения* между типами, подставляемыми вместо данной переменной.
- Для этого предполагаем теоретико-множественную интерпретацию системы типов (т.е. интерпретацию в категории SET).

Отношения вместо функций

- Вместо функций («стрелок») будем рассматривать *отношения* между типами, подставляемыми вместо данной переменной.
- Для этого предполагаем теоретико-множественную интерпретацию системы типов (т.е. интерпретацию в категории SET).
    - Здесь мы ограничиваемся системой типов Хиндли – Милнера: бескванторный тип T интерпретируется множеством, а терму $u : \forall r_1 \ldots r_n.T$ соответствует семейство элементов $u_{r_1 \ldots r_n}$ соответствующих множеств.

Отношения вместо функций

- Вместо функций («стрелок») будем рассматривать *отношения* между типами, подставляемыми вместо данной переменной.
- Для этого предполагаем теоретико-множественную интерпретацию системы типов (т.е. интерпретацию в категории SET).
    - Здесь мы ограничиваемся системой типов Хиндли – Милнера: бескванторный тип T интерпретируется множеством, а терму $u : \forall r_1 \dots r_n.T$ соответствует семейство элементов $u_{r_1 \dots r_n}$ соответствующих множеств.
    - Например, $\mathbf{B}_{\mathtt{Int},\mathtt{Int},\mathtt{Int}}$ — композиция целочисленных функций.

Отношения вместо функций

- Вместо функций («стрелок») будем рассматривать *отношения* между типами, подставляемыми вместо данной переменной.
- Для этого предполагаем теоретико-множественную интерпретацию системы типов (т.е. интерпретацию в категории SET).
    - Здесь мы ограничиваемся системой типов Хиндли – Милнера: бескванторный тип T интерпретируется множеством, а терму $u : \forall r_1 \dots r_n.T$ соответствует семейство элементов $u_{r_1 \dots r_n}$ соответствующих множеств.
    - Например, $\mathbf{B}_{\mathtt{Int},\mathtt{Int},\mathtt{Int}}$ — композиция целочисленных функций.
    - В общем случае системы $\lambda 2$ у нас нет теоретико-множественной интерпретации для $\forall$, и пришлось бы использовать более сложную семантику.

Отношения

- Бинарным отношением (точнее, бинарным соответствием) между множествами A и A' называется подмножество $R \subseteq A \times A'$.

Отношения

- Бинарным отношением (точнее, бинарным соответствием) между множествами A и A' называется подмножество $R \subseteq A \times A'$.
    - Частный случай — функция $f : A \to A'$, где $R_f = \{(x, f(x)) \mid x \in A\}$.

Отношения

- Бинарным отношением (точнее, бинарным соответствием) между множествами A и A' называется подмножество $R \subseteq A \times A'$.
    - Частный случай — функция $f\colon A \to A'$, где $R_f = \{(x, f(x)) \mid x \in A\}$.
    - Мы будем считать «похожими» элементы, находящиеся в отношении.

Отношения

- Бинарным отношением (точнее, бинарным соответствием) между множествами A и A' называется подмножество $R \subseteq A \times A'$.
  - Частный случай — функция $f : A \to A'$, где $R_f = \{(x, f(x)) \mid x \in A\}$.
  - Мы будем считать «похожими» элементы, находящиеся в отношении.
- Пусть дан полиморфный тип $\forall p_1 \dots p_n . T(p_1, \dots, p_n)$, где T бескванторный, и даны отношения $R_1 \subseteq A_1 \times A_1'$, ..., $R_n \subseteq A_n \times A_n'$.

Отношения

- Бинарным отношением (точнее, бинарным соответствием) между множествами A и A' называется подмножество $R \subseteq A \times A'$.
    - Частный случай — функция $f : A \to A'$, где $R_f = \{(x, f(x)) \mid x \in A\}$.
    - Мы будем считать «похожими» элементы, находящиеся в отношении.
- Пусть дан полиморфный тип $\forall p_1 \dots p_n.T(p_1, \dots, p_n)$, где T бескванторный, и даны отношения $R_1 \subseteq A_1 \times A'_1$, ..., $R_n \subseteq A_n \times A'_n$.
- Определим отношение $R \subseteq T(A_1, \dots, A_n) \times T(A'_1, \dots, A'_n)$.

Отношения

- Отношение R определяется рекурсивно. Если $T = p_i$, то $R = R_i \subseteq A_i \times A_i'$.

Отношения

- Отношение R определяется рекурсивно. Если $T = p_i$, то $R = R_i \subseteq A_i \times A_i'$.
- Если $T = T_1 \to T_2$ и $f : T_1(A_1, \dots, A_n) \to T_2(A_1, \dots, A_n)$, а $f' : T_1(A_1', \dots, A_n') \to T_2(A_1', \dots, A_n')$, положим $f R f'$ тогда и только тогда, когда для любого a, a', если $a R a'$, то $f(a) R f'(a')$.

Отношения

- Отношение R определяется рекурсивно. Если $T = p_i$, то $R = R_i \subseteq A_i \times A_i'$.
- Если $T = T_1 \to T_2$ и $f : T_1(A_1, \dots, A_n) \to T_2(A_1, \dots, A_n)$, а $f' : T_1(A_1', \dots, A_n') \to T_2(A_1', \dots, A_n')$, положим $f R f'$ тогда и только тогда, когда для любого a, a', если aRa', то $f(a) R f'(a')$.
- Можно распространить это определение на другие конструкторы типов, например, для списков $[c_1, \dots, c_n]\, R\, [c_1', \dots, c_n']$ тогда и только тогда, когда $c_i R c_i'$.

Отношения

- Отношение R определяется рекурсивно. Если $T = p_i$, то $R = R_i \subseteq A_i \times A_i'$.
- Если $T = T_1 \to T_2$ и $f : T_1(A_1, \dots, A_n) \to T_2(A_1, \dots, A_n)$, а $f' : T_1(A_1', \dots, A_n') \to T_2(A_1', \dots, A_n')$, положим fRf' тогда и только тогда, когда для любого a, a', если aRa', то $f(a)Rf'(a')$.
- Можно распространить это определение на другие конструкторы типов, например, для списков $[c_1, \dots, c_n]\, R\, [c_1', \dots, c_n']$ тогда и только тогда, когда $c_i R c_i'$.
    - В частности, всегда $[]\, R\, []$.

Теорема о полиморфизме

Теорема

Пусть u *— терм типа* $\forall p_1 \dots p_n.T$ *без свободных переменных, то* $u_{A_1,\dots,A_n} \; R \; u_{A'_1,\dots,A'_n}$ *для* ***произвольных*** $R_1, \dots, R_n$.

Теорема о полиморфизме

Теорема

Пусть u *— терм типа* $\forall p_1 \dots p_n.T$ *без свободных переменных, то* $u_{A_1,\dots,A_n} \; R \; u_{A'_1,\dots,A'_n}$ *для* ***произвольных*** $R_1, \dots, R_n$.

- Смысл этой теоремы: разные реализации полиморфного терма всегда «похожи» друг на друга.

Теорема о полиморфизме

Теорема

Пусть u *— терм типа* $\forall p_1 \dots p_n . T$ *без свободных переменных, то* $u_{A_1,\dots,A_n} \; R \; u_{A'_1,\dots,A'_n}$ *для* ***произвольных*** $R_1, \dots, R_n$.

- Смысл этой теоремы: разные реализации полиморфного терма всегда «похожи» друг на друга.
- Пример: $u : \forall p.(p \to p)$. Докажем, что u реализует тождественную функцию.

Теорема о полиморфизме

Теорема

Пусть u *— терм типа* $\forall p_1 \dots p_n.T$ *без свободных переменных, то* $u_{A_1,\dots,A_n}\ R\ u_{A'_1,\dots,A'_n}$ *для* ***произвольных*** $R_1, \dots, R_n$.

- Смысл этой теоремы: разные реализации полиморфного терма всегда «похожи» друг на друга.
- Пример: $u : \ \forall p.(p \to p)$. Докажем, что u реализует тождественную функцию.
- Пусть $u_A : \ A \to A$ и $u_A(b) = d \neq b$. Зададим на $A \times A$ отношение $aRa' \iff a' = b$.

Теорема о полиморфизме

Теорема

Пусть u — терм типа $\forall p_1 \dots p_n.T$ без свободных переменных, то $u_{A_1,\dots,A_n} \, R \, u_{A'_1,\dots,A'_n}$ для ***произвольных*** *$R_1, \dots, R_n$.*

- Смысл этой теоремы: разные реализации полиморфного терма всегда «похожи» друг на друга.
- Пример: $u : \ \forall p.(p \to p)$. Докажем, что u реализует тождественную функцию.
- Пусть $u_A : \ A \to A$ и $u_A(b) = d \neq b$. Зададим на $A \times A$ отношение $aRa' \iff a' = b$.
- Тогда bRb, но неверно, что $u_A(b) \, R \, u_A(b)$. Противоречие с $u_A \, R \, u_A$.

Теорема о полиморфизме

- Чтобы доказать теорему о полиморфизме, нужно её немного обобщить, разрешив свободные переменные $x_1 : T_1, \ldots, x_n : T_n$ и добавив условия $c_1 R c_1', \ldots, c_n R c_n'$, где c_i и c_i' — элементы, подставляемые вместо x_i.

Теорема о полиморфизме

- Чтобы доказать теорему о полиморфизме, нужно её немного обобщить, разрешив свободные переменные $x_1 : T_1, \ldots, x_n : T_n$ и добавив условия $c_1 R c_1', \ldots, c_n R c_n'$, где c_i и c_i' — элементы, подставляемые вместо x_i.
- В таком виде теорема доказывается индукцией по построению u.

Теорема о полиморфизме

- Чтобы доказать теорему о полиморфизме, нужно её немного обобщить, разрешив свободные переменные $x_1 : T_1, \ldots, x_n : T_n$ и добавив условия $c_1 R c_1', \ldots, c_n R c_n'$, где c_i и c_i' — элементы, подставляемые вместо x_i.
- В таком виде теорема доказывается индукцией по построению u.
- Если $u = vw$, то wRw' и vRv', поэтому uRu'.

Теорема о полиморфизме

- Чтобы доказать теорему о полиморфизме, нужно её немного обобщить, разрешив свободные переменные $x_1 : T_1, \ldots, x_n : T_n$ и добавив условия $c_1 R c_1', \ldots, c_n R c_n'$, где c_i и c_i' — элементы, подставляемые вместо x_i.
- В таком виде теорема доказывается индукцией по построению u.
- Если $u = vw$, то wRw' и vRv', поэтому uRu'.
- Если $u = (\lambda x.v) : (T_1 \to T_2)$, то действуем так: возьмём a и a', где $a \in T_1$, $a' \in T_1'$, aRa'. Тогда по предположению индукции vRv'.

Композиция

- Пусть $v : \forall pqr.((q \to r) \to (p \to q) \to p \to r)$, докажем, что это комбинатор композиции.
- Предположим противное: для каких-то типов A, B, C и элементов $g : A \to B$, $f : B \to C$ и $z \in A$ имеем $vfgz \neq f(gz)$.
- Определим отношения R_A, R_B, R_C на множествах A, B, C: $aRa' \iff a' = z; bRb' \iff b' = gz; cRc' \iff c' = f(gz)$.
- Тогда zRz, gRg и fRf, но неверно, что $(vfgz)\,R\,(vfgz)$. Противоречие.
- В силу полноты $\beta\eta$-исчисления в категории SET получаем $v =_{\beta\eta} \mathbf{B} = \lambda fgx.f(gx)$.

Выбор

- Более сложный пример. $w : \forall p.(p \to (p \to p))$.

Выбор

- Более сложный пример. $w : \forall p.(p \to (p \to p))$.
- Здесь есть две существенно разные функции: $w_1 = \lambda xy.x$ и $w_2 = \lambda xy.y$.

Выбор

- Более сложный пример. $w : \forall p.(p \to (p \to p))$.
- Здесь есть две существенно разные функции: $w_1 = \lambda xy.x$ и $w_2 = \lambda xy.y$.
- Есть ли ещё?

Выбор

- Более сложный пример. $w : \forall p.(p \to (p \to p))$.
- Здесь есть две существенно разные функции: $w_1 = \lambda xy.x$ и $w_2 = \lambda xy.y$.
- Есть ли ещё?
- Докажем, что нет. Сначала пусть есть такие b_1 и b_2, что $wb_1b_2 = d \notin \{b_1, b_2\}$. Тогда рассмотрим отношение $aRa' \iff (a' = b_1$ или $a' = b_2)$ и получим противоречие.

Выбор

- Более сложный пример. $w : \forall p.(p \to (p \to p))$.
- Здесь есть две существенно разные функции: $w_1 = \lambda xy.x$ и $w_2 = \lambda xy.y$.
- Есть ли ещё?
- Докажем, что нет. Сначала пусть есть такие b_1 и b_2, что $wb_1b_2 = d \notin \{b_1, b_2\}$. Тогда рассмотрим отношение $aRa' \iff (a' = b_1$ или $a' = b_2)$ и получим противоречие.
- В частности, всегда $wbb = b$.

Выбор

- Более сложный пример. $w : \forall p.(p \to (p \to p))$.
- Здесь есть две существенно разные функции: $w_1 = \lambda xy.x$ и $w_2 = \lambda xy.y$.
- Есть ли ещё?
- Докажем, что нет. Сначала пусть есть такие b_1 и b_2, что $wb_1b_2 = d \notin \{b_1, b_2\}$. Тогда рассмотрим отношение $aRa' \iff (a' = b_1 \text{ или } a' = b_2)$ и получим противоречие.
- В частности, всегда $wbb = b$.
- Но возможна ситуация, когда $wb_1b_2 = b_1$, а $wc_1c_2 = c_2$, причём $b_1 \neq b_2$ и $c_1 \neq c_2$.

Выбор

- Более сложный пример. $w : \forall p.(p \to (p \to p))$.
- Здесь есть две существенно разные функции: $w_1 = \lambda xy.x$ и $w_2 = \lambda xy.y$.
- Есть ли ещё?
- Докажем, что нет. Сначала пусть есть такие b_1 и b_2, что $wb_1b_2 = d \notin \{b_1, b_2\}$. Тогда рассмотрим отношение $aRa' \iff (a' = b_1$ или $a' = b_2)$ и получим противоречие.
- В частности, всегда $wbb = b$.
- Но возможна ситуация, когда $wb_1b_2 = b_1$, а $wc_1c_2 = c_2$, причём $b_1 \neq b_2$ и $c_1 \neq c_2$.
- Положим b_1Rc_1 и b_2Rc_2. Противоречие: $b_1 = wb_1b_2 \; R \; wc_1c_2 = c_2$, что не так.

Задача

- Сколько существует различных (с точностью до $\beta\eta$-эквивалентности) термов типа $\forall pq.((p \to p \to q) \to (p \to p \to p \to q))$?

Free Theorems

```haskell
reverse :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
concat :: [[a]] -> [a]
fst :: (a,b) -> a
filter :: (a -> Bool) -> [a] -> [a]

k :: a -> b -> a
id :: a -> a
```

```haskell
reverse . (map f) = (map f) . reverse
map f (x ++ y) = (map f x) ++ (map f y)
(map f) . concat = concat . (map (map f))
fst (x,y) = x
(map f) . (filter (p.f)) =
    (filter p) . (map f)
f (k x y) = k (f x) (g y)
f . id = id . f
```

Неподвижные точки

- Рекурсию можно реализовать через комбинатор неподвижной точки $\mathbb{Y} : \forall p.((p \to p) \to p)$.

Неподвижные точки

- Рекурсию можно реализовать через комбинатор неподвижной точки $\mathbb{Y} : \forall p.((p \to p) \to p)$.
- Если $f_1 \mathrel{R} f_2$, то **если вычисления завершаются,** имеем $(\mathbb{Y}\, f_1) \mathrel{R} (\mathbb{Y}\, f_2)$.

Неподвижные точки

- Рекурсию можно реализовать через комбинатор неподвижной точки $\mathbb{Y} : \forall p.((p \to p) \to p)$.
- Если $f_1 \, R \, f_2$, то **если вычисления завершаются,** имеем $(\mathbb{Y}\, f_1)\, R\, (\mathbb{Y}\, f_2)$.
    - Действительно, $\mathbb{Y}\, f_i \to f_i(\mathbb{Y}\, f_i)$, и далее действуем по индукции по длине цепочки редукций.

Неподвижные точки

- Рекурсию можно реализовать через комбинатор неподвижной точки $\mathbb{Y} : \forall p.((p \to p) \to p)$.
- Если $f_1 \, R \, f_2$, то **если вычисления завершаются,** имеем $(\mathbb{Y} f_1) \, R \, (\mathbb{Y} f_2)$.
  - Действительно, $\mathbb{Y} f_i \to f_i(\mathbb{Y} f_i)$, и далее действуем по индукции по длине цепочки редукций.
- Поэтому в присутствии $\mathbb{Y}$-комбинатора мы получаем те же свободные теоремы *в слабом смысле:* если обе части вычисляются (на данном входе) за конечное время, то результат одинаков.

Неподвижные точки

- Убрать это условие нельзя.

Неподвижные точки

- Убрать это условие нельзя.
- Действительно, с помощью $\mathbb{Y}$ можно определить константу $\mathbb{Y}(\lambda x.x) : \forall p.p$.

- Убрать это условие нельзя.
- Действительно, с помощью $\mathbb{Y}$ можно определить константу $\mathbb{Y}(\lambda x.x) : \forall p.p$.
- Значит, $f = \lambda z.(\mathbb{Y}(\lambda x.x))$ можно присвоить тип $\forall p.(p \to p)$.

Неподвижные точки

- Убрать это условие нельзя.
- Действительно, с помощью $\mathbb{Y}$ можно определить константу $\mathbb{Y}(\lambda x.x) : \forall p.p$.
- Значит, $f = \lambda z.(\mathbb{Y}(\lambda x.x))$ можно присвоить тип $\forall p.(p \to p)$.
  - Наиболее общий тип: $\forall pq.(p \to q)$.

Неподвижные точки

- Убрать это условие нельзя.
- Действительно, с помощью $\mathbb{Y}$ можно определить константу $\mathbb{Y}(\lambda x.x) : \forall p.p$.
- Значит, $f = \lambda z.(\mathbb{Y}(\lambda x.x))$ можно присвоить тип $\forall p.(p \to p)$.
    - Наиболее общий тип: $\forall pq.(p \to q)$.
- Однако это не тождественная функция $\mathrm{id} = \lambda x.x$.

Неподвижные точки

- Убрать это условие нельзя.
- Действительно, с помощью $\mathbb{Y}$ можно определить константу $\mathbb{Y}(\lambda x.x) : \forall p.p$.
- Значит, $f = \lambda z.(\mathbb{Y}(\lambda x.x))$ можно присвоить тип $\forall p.(p \to p)$.
    - Наиболее общий тип: $\forall pq.(p \to q)$.
- Однако это не тождественная функция $\mathrm{id} = \lambda x.x$.
- С соотношением вида `f.g = g.f` ситуация более тонкая.

Неподвижные точки

- Убрать это условие нельзя.
- Действительно, с помощью $\mathbb{Y}$ можно определить константу $\mathbb{Y}(\lambda x.x) : \forall p.p$.
- Значит, $f = \lambda z.(\mathbb{Y}(\lambda x.x))$ можно присвоить тип $\forall p.(p \to p)$.
    - Наиболее общий тип: $\forall pq.(p \to q)$.
- Однако это не тождественная функция $\mathrm{id} = \lambda x.x$.
- С соотношением вида `f.g = g.f` ситуация более тонкая.
    - При ретивом порядке вычислений и слева, и справа получим бесконечный цикл.

Неподвижные точки

- Убрать это условие нельзя.
- Действительно, с помощью $\mathbb{Y}$ можно определить константу $\mathbb{Y}(\lambda x.x) : \forall p.p$.
- Значит, $f = \lambda z.(\mathbb{Y}(\lambda x.x))$ можно присвоить тип $\forall p.(p \to p)$.
  - Наиболее общий тип: $\forall pq.(p \to q)$.
- Однако это не тождественная функция $\mathrm{id} = \lambda x.x$.
- С соотношением вида `f.g = g.f` ситуация более тонкая.
  - При ретивом порядке вычислений и слева, и справа получим бесконечный цикл.
  - Однако в ленивом порядке (как в Haskell'е) для `g = \x -> 0` имеем `(g.f) 1 = 0`, а `(f.g) 1` зацикливается.

Неопределённость

- Вместо $\mathbb{Y}$id можно взять `undefined`.

Неопределённость

- Вместо $\mathbb{Y}$id можно взять `undefined`.
- То же происходит в случае списков.

Неопределённость

- Вместо $\mathbb{Y}$id можно взять `undefined`.
- То же происходит в случае списков.
- Пусть `r :: [a] -> [a]`

Неопределённость

- Вместо $\mathbb{Y}$id можно взять `undefined`.
- То же происходит в случае списков.
- Пусть `r :: [a] -> [a]`
- Равенство `(map f).r = r.(map f)` можно «сломать», взяв

```
r = (map (\x -> undefined)) :: [a]->[a]
f = \y -> 0
```

Неопределённость

- Вместо $\mathbb{Y}$id можно взять undefined.
- То же происходит в случае списков.
- Пусть `r :: [a] -> [a]`
- Равенство `(map f).r = r.(map f)` можно «сломать», взяв

  ```
  r = (map (\x -> undefined)) :: [a]->[a]
  f = \y -> 0
  ```

- Здесь `((map f). r) [1,2,3]` вернёт `[0,0,0]`, а вот `(r . (map f)) [1,2,3]` выдаст исключение.

Free Theorems

- Проблемы с неопределённостью в наших свободных теоремах можно исправить, потребовав, чтобы функции f, g были *строгими* в смысле вычисления: $f \bot = \bot$, где $\bot$ — это undefined или незавершающееся вычисление.

Free Theorems

- Проблемы с неопределённостью в наших свободных теоремах можно исправить, потребовав, чтобы функции f, g были *строгими* в смысле вычисления: $f \perp = \perp$, где $\perp$ — это undefined или незавершающееся вычисление.
- Это свойство выполнено при аппликативном порядке вычислений, но не при нормальном.

Free Theorems

- Проблемы с неопределённостью в наших свободных теоремах можно исправить, потребовав, чтобы функции `f`, `g` были *строгими* в смысле вычисления: $f\bot = \bot$, где $\bot$ — это undefined или незавершающееся вычисление.
- Это свойство выполнено при аппликативном порядке вычислений, но не при нормальном.
- При этих условиях можно добавить $\bot$ в наши отношения, при этом будет только $\bot R \bot$, но не $aR\bot$.

Free Theorems

- Проблемы с неопределённостью в наших свободных теоремах можно исправить, потребовав, чтобы функции f, g были *строгими* в смысле вычисления: $f\bot = \bot$, где $\bot$ — это undefined или незавершающееся вычисление.
- Это свойство выполнено при аппликативном порядке вычислений, но не при нормальном.
- При этих условиях можно добавить $\bot$ в наши отношения, при этом будет только $\bot R \bot$, но не $aR\bot$.
- Свободные теоремы — ещё один пример использования системы типов для частичной *верификации* программного кода: правильно расставленные типы позволяют установить, без анализа кода, некоторые свойства функций.