Функциональное программирование

Лекция 6

Степан Львович Кузнецов

НИУ ВШЭ, факультет компьютерных наук

Функторы

- Функторы и, как частный случай, монады — это *преобразователи типов.* В Haskell'е реализуются как параметрические классы типов.

Функторы

- Функторы и, как частный случай, монады — это *преобразователи типов*. В Haskell'е реализуются как параметрические классы типов.
- Функтор позволяет «поднимать», с помощью `fmap`, функции `a -> b` до `f a -> f b` (математически: $h : X \to Y$ преобразуется в $Fh : FX \to FY$).

Функторы

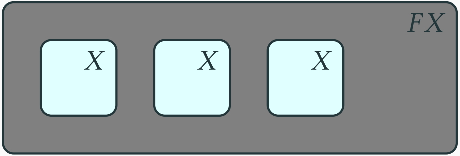
- Функторы и, как частный случай, монады — это *преобразователи типов*. В Haskell'е реализуются как параметрические классы типов.
- Функтор позволяет «поднимать», с помощью `fmap`, функции `a -> b` до `f a -> f b` (математически: $h : X \to Y$ преобразуется в $Fh : FX \to FY$).
  - При этом `fmap` согласована с композицией и тождественным отображением.

Функторы

- Функторы и, как частный случай, монады — это *преобразователи типов*. В Haskell'е реализуются как параметрические классы типов.
- Функтор позволяет «поднимать», с помощью `fmap`, функции `a -> b` до `f a -> f b` (математически: $h: X \to Y$ преобразуется в $Fh: FX \to FY$).
    - При этом `fmap` согласована с композицией и тождественным отображением.
    - Проверка этого на совести автора реализации!

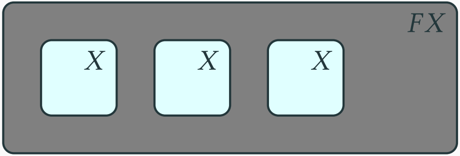
«Чёрный ящик»

- Таким образом, если X — тип, а F — функтор, то FX — это «чёрный ящик», в котором каким-то образом спрятаны элементы типа X, к которым можно применить функцию $f: X \to Y$.



«Чёрный ящик»

- Таким образом, если X — тип, а F — функтор, то FX — это «чёрный ящик», в котором каким-то образом спрятаны элементы типа X, к которым можно применить функцию $f\colon X \to Y$.



- Однако чего-то не хватает: непонятно, как «положить» что-то в FX, и вообще, как создать объект типа FX.

Монада

- *Монада* M — это эндофунктор с дополнительными операциями.

Монада

- *Монада* M — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.

Монада

- *Монада M* — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.
  - `return :: Monad m => a -> m a`

Монада

- *Монада M* — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.
  - `return :: Monad m => a -> m a`
- Вторая более сложная и позволяет «поднимать» аргумент функции, ведущей в монаду. Для $f : X \to MY$ эта операция даёт $f^* : MX \to MY$.

Монада

- *Монада* M — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.
    - `return :: Monad m => a -> m a`
- Вторая более сложная и позволяет «поднимать» аргумент функции, ведущей в монаду. Для $f : X \to MY$ эта операция даёт $f^* : MX \to MY$.
    - `(>>=) :: Monad m => m a -> (a -> m b) -> m b`

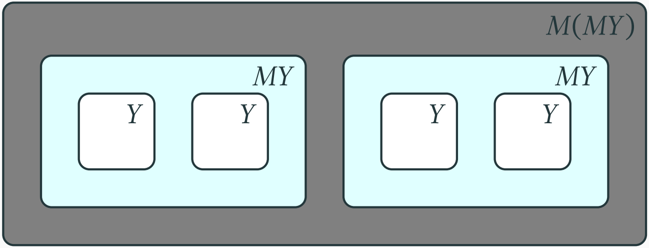
Монада

- *Монада* M — это эндофунктор с дополнительными операциями.
- Первая из них позволяет «положить элемент в чёрный ящик», $\eta_X : X \to MX$.
    - `return :: Monad m => a -> m a`
- Вторая более сложная и позволяет «поднимать» аргумент функции, ведущей в монаду. Для $f : X \to MY$ эта операция даёт $f^* : MX \to MY$.
    - `(>>=) :: Monad m => m a -> (a -> m b) -> m b`
- Набор $(M, \eta, ^*)$ в теории категорий называется *тройкой Клейсли*, а собственно монадой — другая, но эквивалентная конструкция.

- Операция *, или `>>=`, соответствует идее, что можно внутри монады войти ещё раз в монаду, и остаться в той же монаде.

- Операция *, или `>>=`, соответствует идее, что можно внутри монады войти ещё раз в монаду, и остаться в той же монаде.
- Пример: `[1,2,3] >>= (\x -> [x,x])` даёт `[1,1,2,2,3,3]`

- Операция *, или `>>=`, соответствует идее, что можно внутри монады войти ещё раз в монаду, и остаться в той же монаде.
- Пример: `[1,2,3] >>= (\x -> [x,x])` даёт `[1,1,2,2,3,3]`
- Если бы вместо f^* применили M как функтор, то получилось бы $Mf : MX \to M(MY)$.



- Таким образом, f^* можно свести к более простой операции «разглаживания» двойной монады в одинарную, $\mu_X : M(MY) \to MY$.

- Таким образом, f^* можно свести к более простой операции «разглаживания» двойной монады в одинарную, $\mu_X : M(MY) \to MY$.
- Тогда $f^* = \mu \circ (Mf)$.

- Таким образом, f^* можно свести к более простой операции «разглаживания» двойной монады в одинарную, $\mu_X : M(MY) \to MY$.
- Тогда $f^* = \mu \circ (Mf)$.
- Собственно, в теории категорий именно эндофунктор M, оснащённый семействами морфизмов η и μ, удовлетворяющий огромному количеству условий корректности, и называют монадой.

Условия монады

Для тройки Клейсли $(M, \eta, {}^*)$ условия формулируются намного короче:

$\eta^*_X = \mathbf{1}_{MX}$

```
(z >>= return) = z
```

$MX \to MX$

$f^* \circ \eta_X = f$

```
(return x >>= f) = (f x)
```

$X \xrightarrow{\eta} MX \xrightarrow{f^*} MY$

$g^* \circ f^* = (g^* \circ f)^*$

```
((x >>= f) >>= g) =
    (x >>= (\y -> (f y >>= g)))
```

$MX \xrightarrow{f^*} MY \xrightarrow{g^*} MZ$

$X \xrightarrow{f} MY \xrightarrow{g^*} MZ$

Монада IO

- Одна из наиболее известных монад — это монада `IO`, с помощью которой реализуется «выход во внешний мир».

Монада IO

- Одна из наиболее известных монад — это монада `IO`, с помощью которой реализуется «выход во внешний мир».
- Объект типа `IO a` можно понимать как объект типа a, помещённый в большой и страшный внешний мир.

Монада IO

- Одна из наиболее известных монад — это монада `IO`, с помощью которой реализуется «выход во внешний мир».
- Объект типа `IO a` можно понимать как объект типа `a`, помещённый в большой и страшный внешний мир.
- При этом с IO-монадическими объектами можно работать «чисто функциональным» образом (они существуют как задумки, а реализуются только при финальном вычислении).

Монада IO

- Одна из наиболее известных монад — это монада `IO`, с помощью которой реализуется «выход во внешний мир».
- Объект типа `IO a` можно понимать как объект типа `a`, помещённый в большой и страшный внешний мир.
- При этом с IO-монадическими объектами можно работать «чисто функциональным» образом (они существуют как задумки, а реализуются только при финальном вычислении).
- Однако сегодня мы поговорим о монадах, которые упрощают чисто функциональное программирование.

do-нотация

- *do-нотация* — это альтернативный синтаксис работы с `>>=` и `>>`, делающий код похожим на императивный.

do-нотация

- *do-нотация* — это альтернативный синтаксис работы с `>>=` и `>>`, делающий код похожим на императивный.
- Пример:

```
main :: IO ()
main = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

do-нотация

- *do-нотация* — это альтернативный синтаксис работы с `>>=` и `>>`, делающий код похожим на императивный.
- Пример:

```haskell
main :: IO ()
main = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name ++ "!"
```

- do-нотация раскрывается так. «Команды», у которых нет возвращаемого значения, соединяются с помощью `>>`. Если возвращаемое значение есть: `x <- ...`, то пишется `... >>= \x -> ...`

do-нотация

- Таким образом переменная по имени x становится доступной в дальнейшем контексте.

do-нотация

- Таким образом переменная по имени x становится доступной в дальнейшем контексте.
- Более того, в «присваивании» `<-` можно (как в императивных языках) использовать одно и то же имя несколько раз.

do-нотация

- Таким образом переменная по имени x становится доступной в дальнейшем контексте.
- Более того, в «присваивании» `<-` можно (как в императивных языках) использовать одно и то же имя несколько раз.
    - При этом более раннее забывается, поскольку переменная связана более глубокой лямбдой: $\lambda x.(\ldots\lambda x.(\ldots x\ldots)\ldots)$.

do-нотация

- Таким образом переменная по имени x становится доступной в дальнейшем контексте.
- Более того, в «присваивании» `<-` можно (как в императивных языках) использовать одно и то же имя несколько раз.
  - При этом более раннее забывается, поскольку переменная связана более глубокой лямбдой: $\lambda x.(\ldots\lambda x.(\ldots x\ldots)\ldots)$.
- Пример:

```haskell
main =
  putStrLn "What's your name?" >>
  getLine >>=
  \name -> putStrLn $ "Hello, " ++ name ++ "!"
```

do-нотация

- do-нотация работает не только с `IO`, но и с любой другой монадой.

do-нотация

- do-нотация работает не только с `IO`, но и с любой другой монадой.
- Например, вот такой код рекурсивно генерирует все булевы наборы данной длины:

```
allVals 0 = [[]]
allVals n = do
  v <- allVals (n-1)
  [True:v, False:v]
```

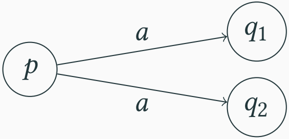
Монада для недетерминизма

- В первом примере мы будем моделировать поведение *недетерминированного конечного автомата (НКА).*

Монада для недетерминизма

- В первом примере мы будем моделировать поведение *недетерминированного конечного автомата (НКА).*
- Напомним, что конечный автомат имеет конечное множество *состояний* Q и перемещается между ними в зависимости от очередного символа входного слова w.

- В первом примере мы будем моделировать поведение *недетерминированного конечного автомата (НКА).*
- Напомним, что конечный автомат имеет конечное множество *состояний* Q и перемещается между ними в зависимости от очередного символа входного слова w.
- Автомат недетерминированный, если буква входного слова не обязательно однозначно определяет переход:



Монада для недетерминизма

- Таким образом, после n шагов, т.е. входного слова $w = a_1 \ldots a_n$, мы имеем некоторое подмножество *возможных* состояний автомата, $\Delta^n(w) \subseteq Q$.

- Таким образом, после n шагов, т.е. входного слова $w = a_1 \dots a_n$, мы имеем некоторое подмножество *возможных* состояний автомата, $\Delta^n(w) \subseteq Q$.
- При этом сама функция перехода действует из состояния во множество состояний, $\delta : \Sigma \to (Q \to \mathscr{P}Q)$.

Монада для недетерминизма

- Таким образом, после n шагов, т.е. входного слова $w = a_1 \dots a_n$, мы имеем некоторое подмножество *возможных* состояний автомата, $\Delta^n(w) \subseteq Q$.
- При этом сама функция перехода действует из состояния во множество состояний, $\delta : \Sigma \to (Q \to \mathscr{P}Q)$.
- Это в точности ситуация *монадического связывания* для монады $\mathscr{P}$:

$$\Delta(w) = (\eta\, q_0) \mathbin{>\!\!>\!=} \delta(a_1) \mathbin{>\!\!>\!=} \dots \mathbin{>\!\!>\!=} \delta(a_n)$$

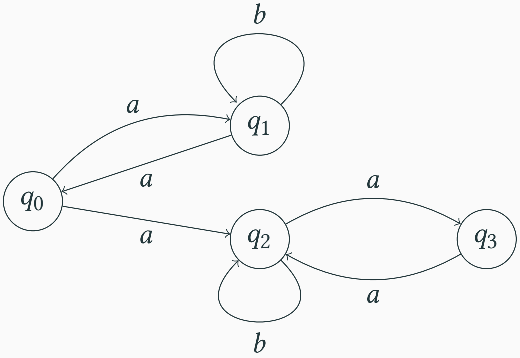
Монада для недетерминизма

- Таким образом, после n шагов, т.е. входного слова $w = a_1 \dots a_n$, мы имеем некоторое подмножество *возможных* состояний автомата, $\Delta^n(w) \subseteq Q$.
- При этом сама функция перехода действует из состояния во множество состояний, $\delta : \Sigma \to (Q \to \mathscr{P}Q)$.
- Это в точности ситуация *монадического связывания* для монады $\mathscr{P}$:

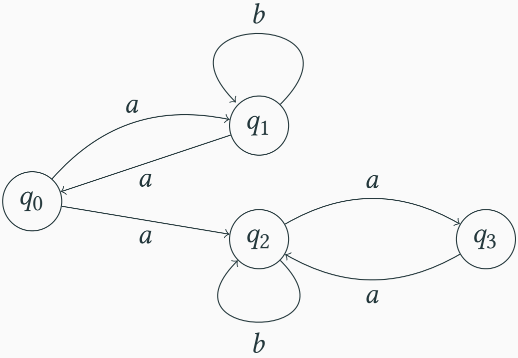
$$\Delta(w) = (\eta\, q_0) \mathbin{\texttt{>>=}} \delta(a_1) \mathbin{\texttt{>>=}} \dots \mathbin{\texttt{>>=}} \delta(a_n)$$

  - Здесь $\eta\, q_0 = \texttt{return}\ \ q_0 = \{q_0\}$.

Пример конечного автомата



Пример конечного автомата



- На входном слове из $(ab^*a)^*$ этот автомат переходит либо в q_0, либо в q_3.

Монада для недетерминизма

Идея монады $\mathscr{P}$ для недетерминизма в точности реализуется:

```haskell
import Control.Monad
import Data.Set.Monad

data NFA q sigma = NFA (sigma -> q -> Set q) q

runq :: NFA q sigma -> q -> [sigma] -> Set q
runq (NFA delta qi) q0 [] = return q0
runq (NFA delta qi) q0 (a:ss) = do
    q1 <- delta a q0
    q2 <- runq (NFA delta qi) q1 ss
    return q2

run (NFA delta qi) a = runq (NFA delta qi) qi a
```

Монада для недетерминизма

Далее, можно закодировать автомат из нашего примера:

```
data Q = Q0 | Q1 | Q2 | Q3 deriving (Eq, Ord, Show)

delta :: Char -> Q -> Set Q
delta 'a' Q0 = fromList [Q1,Q2]
delta 'b' Q0 = fromList []
delta 'a' Q1 = fromList [Q0]
delta 'b' Q1 = fromList [Q1]
delta 'a' Q2 = fromList [Q3]
delta 'b' Q2 = fromList [Q2]
delta 'a' Q3 = fromList [Q2]
delta 'b' Q3 = fromList []

automaton = NFA delta Q0
```

Монада для недетерминизма

- Наконец, запускаем:

  ```
  main = putStrLn $ show $ run automaton "abbaabbba"
  ```

  и получаем ответ: `fromList [Q0,Q3]`.

Монада для недетерминизма

- Наконец, запускаем:

  ```
  main = putStrLn $ show $ run automaton "abbaabbba"
  ```

  и получаем ответ: `fromList [Q0,Q3]`.
- Заметим, что если использовать монаду списка вместо `Set`, то состояния будут дублироваться: `[Q0,Q3,Q3]`.

Монада для недетерминизма

- Наконец, запускаем:
  `main = putStrLn $ show $ run automaton "abbaabbba"`
  и получаем ответ: `fromList [Q0,Q3]`.
- Заметим, что если использовать монаду списка вместо `Set`, то состояния будут дублироваться: `[Q0,Q3,Q3]`.
- Это может привести к экспоненциальному расходу ресурсов.

Монада для недетерминизма

- Наконец, запускаем:
  `main = putStrLn $ show $ run automaton "abbaabbba"`
  и получаем ответ: `fromList [Q0,Q3]`.
- Заметим, что если использовать монаду списка вместо `Set`, то состояния будут дублироваться: `[Q0,Q3,Q3]`.
- Это может привести к экспоненциальному расходу ресурсов.
- Переход от Q к $\mathscr{P}Q$ и замена функции перехода $\delta(a) : Q \to \mathscr{P}Q$ на $\delta(a)^* : \mathscr{P}Q \to \mathscr{P}Q$ — это и есть *алгоритм детерминизации* конечного автомата.

Монада для недетерминизма

- Наконец, запускаем:
  ```
  main = putStrLn $ show $ run automaton "abbaabbba"
  ```
  и получаем ответ: `fromList [Q0,Q3]`.
- Заметим, что если использовать монаду списка вместо `Set`, то состояния будут дублироваться: `[Q0,Q3,Q3]`.
- Это может привести к экспоненциальному расходу ресурсов.
- Переход от Q к $\mathscr{P}Q$ и замена функции перехода $\delta(a) : Q \to \mathscr{P}Q$ на $\delta(a)^* : \mathscr{P}Q \to \mathscr{P}Q$ — это и есть *алгоритм детерминизации* конечного автомата.
- В нашей реализации мы не храним детерминированную версию конечного автомата, тем самым избегая экспоненциального расхода памяти (ленивость!).

Случайные числа

- Функции в Haskell'е должны быть чистыми. Таким образом, мы не можем определить функцию (без аргументов) "random", которая будет при каждом вызове давать новое (псевдо)случайное число.

Случайные числа

- Функции в Haskell'е должны быть чистыми. Таким образом, мы не можем определить функцию (без аргументов) "random", которая будет при каждом вызове давать новое (псевдо)случайное число.
- Однако это возможно внутри монады.

Случайные числа

- Функции в Haskell'е должны быть чистыми. Таким образом, мы не можем определить функцию (без аргументов) “random”, которая будет при каждом вызове давать новое (псевдо)случайное число.
- Однако это возможно внутри монады.
- В частности, в монаде `IO` имеется `randomIO`, который выдаёт псевдослучайное число (в зависимости от конкретизации типа).

Случайные числа

- Функции в Haskell'е должны быть чистыми. Таким образом, мы не можем определить функцию (без аргументов) "random", которая будет при каждом вызове давать новое (псевдо)случайное число.
- Однако это возможно внутри монады.
- В частности, в монаде `IO` имеется `randomIO`, который выдаёт псевдослучайное число (в зависимости от конкретизации типа).
- Однако Haskell поддерживает и более абстрактный способ работы с вероятностными объектами.

Вероятностные распределения

- Вероятностное распределение на конечном множестве $X = \{x_1, \dots, x_n\}$ (дискретное) задаётся набором чисел $(p_1, \dots, p_n)$, где $p_i \geqslant 0$ и $p_1 + \dots + p_n = 1$.

Вероятностные распределения

- Вероятностное распределение на конечном множестве $X = \{x_1, \dots, x_n\}$ (дискретное) задаётся набором чисел $(p_1, \dots, p_n)$, где $p_i \geqslant 0$ и $p_1 + \dots + p_n = 1$.
- $p_i = p(x_i)$.

Вероятностные распределения

- Вероятностное распределение на конечном множестве $X = \{x_1, \dots, x_n\}$ (дискретное) задаётся набором чисел $(p_1, \dots, p_n)$, где $p_i \geqslant 0$ и $p_1 + \dots + p_n = 1$.
- $p_i = p(x_i)$.
- Для события $A \subseteq X$ имеем $P(A) = \sum\limits_{x \in A} p(x)$.

Вероятностные распределения

- Вероятностное распределение на конечном множестве $X = \{x_1, \ldots, x_n\}$ (дискретное) задаётся набором чисел $(p_1, \ldots, p_n)$, где $p_i \geqslant 0$ и $p_1 + \ldots + p_n = 1$.
- $p_i = p(x_i)$.
- Для события $A \subseteq X$ имеем $P(A) = \sum\limits_{x \in A} p(x)$.
- В непрерывном случае вероятностное распределение задано функцией плотности p; $p(x) \geqslant 0$ и $\int\limits_X p(x)\,dx = 1$.

Вероятностные распределения

- Вероятностное распределение на конечном множестве $X = \{x_1, \dots, x_n\}$ (дискретное) задаётся набором чисел $(p_1, \dots, p_n)$, где $p_i \geqslant 0$ и $p_1 + \dots + p_n = 1$.
- $p_i = p(x_i)$.
- Для события $A \subseteq X$ имеем $P(A) = \sum\limits_{x \in A} p(x)$.
- В непрерывном случае вероятностное распределение задано функцией плотности p; $p(x) \geqslant 0$ и $\int\limits_X p(x)\,dx = 1$.
- При этом $P(A) = \int\limits_A p(x)\,dx$.

Монада Жири

- Обозначим через $\mathscr{R}X$ множество всех вероятностных распределений на X.

Монада Жири

- Обозначим через $\mathscr{R}X$ множество всех вероятностных распределений на X.
- $\mathscr{R}$ является функтором. Функция $f : X \to Y$ переносит вероятностное распределение с X на Y следующим образом: $p'(y) = P(f^{-1}(y)) = \sum\limits_{f(x)=y} p(x)$.

Монада Жири

- Обозначим через $\mathscr{R}X$ множество всех вероятностных распределений на X.
- $\mathscr{R}$ является функтором. Функция $f\colon X \to Y$ переносит вероятностное распределение с X на Y следующим образом: $p'(y) = P(f^{-1}(y)) = \sum\limits_{f(x)=y} p(x)$.
    - При этом выполняются условия функтора.

Монада Жири

- Обозначим через $\mathscr{R}X$ множество всех вероятностных распределений на X.
- $\mathscr{R}$ является функтором. Функция $f : X \to Y$ переносит вероятностное распределение с X на Y следующим образом: $p'(y) = P(f^{-1}(y)) = \sum_{f(x)=y} p(x)$.
    - При этом выполняются условия функтора.
- Более того, $\mathscr{R}$ является монадой.

Монада Жири

- Обозначим через $\mathscr{R}X$ множество всех вероятностных распределений на X.
- $\mathscr{R}$ является функтором. Функция $f\colon X \to Y$ переносит вероятностное распределение с X на Y следующим образом: $p'(y) = P(f^{-1}(y)) = \sum_{f(x)=y} p(x)$.
    - При этом выполняются условия функтора.
- Более того, $\mathscr{R}$ является монадой.
- Эта монада называется *монадой Жирú* (Giry monad).

- Обозначим через $\mathscr{R}X$ множество всех вероятностных распределений на X.
- $\mathscr{R}$ является функтором. Функция $f : X \to Y$ переносит вероятностное распределение с X на Y следующим образом: $p'(y) = P(f^{-1}(y)) = \sum\limits_{f(x)=y} p(x)$.
    - При этом выполняются условия функтора.
- Более того, $\mathscr{R}$ является монадой.
- Эта монада называется *монадой Жири́* (Giry monad).
    - M. Giry. A categorical approach to probability theory

- Морфизм $\eta : X \to \mathcal{R}X$ реализуется взятием распределения, сосредоточенного в одной точке:

$$p(y) = \begin{cases} 1, & \text{если } y = x; \\ 0, & \text{иначе.} \end{cases}$$

Монада Жири

- Морфизм $\eta : X \to \mathscr{R}X$ реализуется взятием распределения, сосредоточенного в одной точке:
$$p(y) = \begin{cases} 1, & \text{если } y = x; \\ 0, & \text{иначе.} \end{cases}$$
- Операцию связывания (`>>=`) будем определять через μ (join): если $f : X \to \mathscr{R}Y$, то $(p \mathbin{\texttt{>>=}} f) = \mu \circ (\mathscr{R}f)$.

Монада Жири

- Морфизм $\eta : X \to \mathscr{R}X$ реализуется взятием распределения, сосредоточенного в одной точке:

$$p(y) = \begin{cases} 1, & \text{если } y = x; \\ 0, & \text{иначе.} \end{cases}$$

- Операцию связывания (`>>=`) будем определять через μ (join): если $f : X \to \mathscr{R}Y$, то $(p \mathbin{\texttt{>>=}} f) = \mu \circ (\mathscr{R}f)$.
- Морфизм $\mu : \mathscr{R}\mathscr{R}X \to \mathscr{R}X$ делает обычное распределение из «распределения распределений».

Монада Жири

- Морфизм $\eta : X \to \mathscr{R}X$ реализуется взятием распределения, сосредоточенного в одной точке:

$$p(y) = \begin{cases} 1, & \text{если } y = x; \\ 0, & \text{иначе.} \end{cases}$$

- Операцию связывания (`>>=`) будем определять через μ (join): если $f : X \to \mathscr{R}Y$, то $(p \mathbin{\texttt{>>=}} f) = \mu \circ (\mathscr{R}f)$.
- Морфизм $\mu : \mathscr{R}\mathscr{R}X \to \mathscr{R}X$ делает обычное распределение из «распределения распределений».
- Чтобы получить случайный x по распределению $P \in \mathscr{R}\mathscr{R}X$, мы сначала случайно выбираем распределение $p \in \mathscr{R}X$, а потом по этому распределению выбираем $x \in X$.

Монада Жири

- Морфизм $\eta : X \to \mathscr{R}X$ реализуется взятием распределения, сосредоточенного в одной точке:

$$p(y) = \begin{cases} 1, & \text{если } y = x; \\ 0, & \text{иначе.} \end{cases}$$

- Операцию связывания (>>=) будем определять через μ (join): если $f : X \to \mathscr{R}Y$, то $(p \mathbin{>\!>=} f) = \mu \circ (\mathscr{R}f)$.
- Морфизм $\mu : \mathscr{R}\mathscr{R}X \to \mathscr{R}X$ делает обычное распределение из «распределения распределений».
- Чтобы получить случайный x по распределению $P \in \mathscr{R}\mathscr{R}X$, мы сначала случайно выбираем распределение $p \in \mathscr{R}X$, а потом по этому распределению выбираем $x \in X$.
- $(\mu P)(x) = \sum\limits_{p \in \mathscr{R}X} (P(p) \cdot p(x))$

Монада Жири

- Связывание, $p \mathbin{>\!\!>\!\!=} f$, где $p : \mathscr{R}X$ и $f : X \to \mathscr{R}Y$, получается таким образом:

$$(p \mathbin{>\!\!>\!\!=} f)(y) = \sum_{x \in X} (p(x) \cdot f(x)(y))$$

Монада Жири

- Связывание, $p \mathbin{\texttt{>>=}} f$, где $p : \mathscr{R}X$ и $f : X \to \mathscr{R}Y$, получается таким образом:

$$(p \mathbin{\texttt{>>=}} f)(y) = \sum_{x \in X} (p(x) \cdot f(x)(y))$$

- Вероятностный смысл: выбираем случайный $x \in X$ и по нему строим новое распределение $f(x)$, с помощью которого выбираем $y \in Y$.

Вычисления со случайностью

- Внутри монады $\mathscr{R}$ можно «присваивать» переменной случайное значение (в рамках do-нотации):

```haskell
import Control.Monad.Random
fair = fromList [("heads",0.5),("tails",0.5)]
quart = do
    a <- fair
    b <- fair
    if (a == "heads") && (b == "heads")
       then return "heads"
       else return "tails"
```

Вычисления со случайностью

- Внутри монады $\mathscr{R}$ можно «присваивать» переменной случайное значение (в рамках do-нотации):

```
import Control.Monad.Random
fair = fromList [("heads",0.5),("tails",0.5)]
quart = do
    a <- fair
    b <- fair
    if (a == "heads") && (b == "heads")
       then return "heads"
       else return "tails"
```

- Однако здесь просто *вычисляются* параметры вероятностного распределения, а не *генерируется случайное число.*

Абстрактная работа со случайными объектами

- Посмотрим внимательнее на типы.

```
fromList :: MonadRandom m => [(a, Rational)] -> m a
fair :: MonadRandom m => m [Char]
```

Абстрактная работа со случайными объектами

- Посмотрим внимательнее на типы.

```
fromList :: MonadRandom m => [(a, Rational)] -> m a
fair :: MonadRandom m => m [Char]
```

- Мы видим *полиморфный* объект, тип которого параметризован *произвольной* «монадой случайности» `m` (которая как-то хранит вероятностное распределение).

Абстрактная работа со случайными объектами

- Посмотрим внимательнее на типы.

```
fromList :: MonadRandom m => [(a, Rational)] -> m a
fair :: MonadRandom m => m [Char]
```

- Мы видим *полиморфный* объект, тип которого параметризован *произвольной* «монадой случайности» `m` (которая как-то хранит вероятностное распределение).
- На этом этапе детали реализации `m` не важны, они понадобятся в тот момент, когда мы захотим на самом деле «подбросить монеты».

Реализация MonadRandom

- Стандартная реализация класса `MonadRandom` даётся двупараметрическим типом `Rand g a` при фиксированном первом параметре.

Реализация MonadRandom

- Стандартная реализация класса `MonadRandom` даётся двупараметрическим типом `Rand g a` при фиксированном первом параметре.
- Параметр-тип g, который должен принадлежать классу `RandomGen`, задаёт тип *генератора* (источника) случайности.

Реализация MonadRandom

- Стандартная реализация класса `MonadRandom` даётся двупараметрическим типом `Rand g a` при фиксированном первом параметре.
- Параметр-тип g, который должен принадлежать классу `RandomGen`, задаёт тип *генератора* (источника) случайности.
- Одним из таких типов является `StdGen`.

Реализация MonadRandom

- Стандартная реализация класса `MonadRandom` даётся двупараметрическим типом `Rand g a` при фиксированном первом параметре.
- Параметр-тип g, который должен принадлежать классу `RandomGen`, задаёт тип *генератора* (источника) случайности.
- Одним из таких типов является `StdGen`.
- Из объекта типа `StdGen` (источник случайности) можно извлечь некоторое случайное значение и, кроме того, новый, модифицированный источник.

Реализация MonadRandom

- Стандартная реализация класса `MonadRandom` даётся двупараметрическим типом `Rand g a` при фиксированном первом параметре.
- Параметр-тип g, который должен принадлежать классу `RandomGen`, задаёт тип *генератора* (источника) случайности.
- Одним из таких типов является `StdGen`.
- Из объекта типа `StdGen` (источник случайности) можно извлечь некоторое случайное значение и, кроме того, новый, модифицированный источник.
- Таким образом реализуется последовательность псевдослучайных чисел.

Реализация MonadRandom

- Стандартная реализация класса `MonadRandom` даётся двупараметрическим типом `Rand g a` при фиксированном первом параметре.
- Параметр-тип g, который должен принадлежать классу `RandomGen`, задаёт тип *генератора* (источника) случайности.
- Одним из таких типов является `StdGen`.
- Из объекта типа `StdGen` (источник случайности) можно извлечь некоторое случайное значение и, кроме того, новый, модифицированный источник.
- Таким образом реализуется последовательность псевдослучайных чисел.
- Важно понимать, что сам источник — это константа, и он всегда будет выдавать одно и то же значение.

Реализация MonadRandom

- В нашем примере `quart` имеет полиморфный тип `MonadRandom m => m [Char]`, который может конкретизироваться в `Rand StdGen [Char]`.

Реализация MonadRandom

- В нашем примере `quart` имеет полиморфный тип `MonadRandom m => m [Char]`, который может конкретизироваться в `Rand StdGen [Char]`.
- Имеется функция `evalRand :: Rand g a -> g -> a` (в частности, `evalRand :: Rand StdGen a -> StdGen -> a`), которая выдаёт случайное значение, с данным распределением, используя данный источник случайности.

Реализация MonadRandom

- В нашем примере `quart` имеет полиморфный тип `MonadRandom m => m [Char]`, который может конкретизироваться в `Rand StdGen [Char]`.
- Имеется функция `evalRand :: Rand g a -> g -> a` (в частности, `evalRand :: Rand StdGen a -> StdGen -> a`), которая выдаёт случайное значение, с данным распределением, используя данный источник случайности.
- Чтобы получить новый источник, нужно воспользоваться другой функцией, `runRand :: Rand g a -> g -> (a,g)`

Реализация MonadRandom

- В нашем примере `quart` имеет полиморфный тип `MonadRandom m => m [Char]`, который может конкретизироваться в `Rand StdGen [Char]`.
- Имеется функция `evalRand :: Rand g a -> g -> a` (в частности, `evalRand :: Rand StdGen a -> StdGen -> a`), которая выдаёт случайное значение, с данным распределением, используя данный источник случайности.
- Чтобы получить новый источник, нужно воспользоваться другой функцией, `runRand :: Rand g a -> g -> (a,g)`
- Остаётся последний вопрос — откуда изначально взять источник случайности?

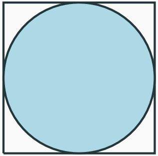
Метод Монте-Карло

- Иногда бывает достаточно использовать фиксированный источник псевдослучайных чисел («зерно»), что позволяет писать чистый и фактически детерминированный код.

Метод Монте-Карло

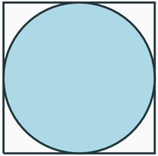
- Иногда бывает достаточно использовать фиксированный источник псевдослучайных чисел («зерно»), что позволяет писать чистый и фактически детерминированный код.
- Одна из таких ситуаций — вычисление меры и интеграла *методом Монте-Карло*.

- Иногда бывает достаточно использовать фиксированный источник псевдослучайных чисел («зерно»), что позволяет писать чистый и фактически детерминированный код.
- Одна из таких ситуаций — вычисление меры и интеграла *методом Монте-Карло.*
- Пример: случайно «бросаем» много точек в квадрат $[0, 1] \times [0, 1]$ и вычисляем долю точек, попавших в круг:



Метод Монте-Карло

- Иногда бывает достаточно использовать фиксированный источник псевдослучайных чисел («зерно»), что позволяет писать чистый и фактически детерминированный код.
- Одна из таких ситуаций — вычисление меры и интеграла *методом Монте-Карло.*
- Пример: случайно «бросаем» много точек в квадрат $[0, 1] \times [0, 1]$ и вычисляем долю точек, попавших в круг:



- Результат по вероятности стремится к площади круга: $\pi/4$.

Метод Монте-Карло

- Для реализации метода Монте-Карло вычисления числа π нам понадобится следующая операция на монадах:

```haskell
replicateM :: Applicative m => Int -> m a -> m [a]
```

Метод Монте-Карло

- Для реализации метода Монте-Карло вычисления числа π нам понадобится следующая операция на монадах:

```haskell
replicateM :: Applicative m => Int -> m a -> m [a]
```

- В частности, для монады replicateM можно определить так:

```haskell
myreplicateM 0 mon = return []
myreplicateM n mon = mon >>= (\x -> myreplicateM (n-1) xs
    >>= \xs -> return (x:xs))
```

или в do-нотации:

```haskell
myreplicateM n mon = do
    x <- mon
    xs <- myreplicateM (n-1) mon
    return (x:xs)
```

Метод Монте-Карло

- Получается, что `replicateM` создаёт *выборку* — список значений случайной величины, причём каждый раз используется обновлённый генератор.

- Получается, что `replicateM` создаёт *выборку* — список значений случайной величины, причём каждый раз используется обновлённый генератор.
- Таким образом, значения получаются псевдонезависимы, что достаточно для статистических целей.

- Получается, что `replicateM` создаёт *выборку* — список значений случайной величины, причём каждый раз используется обновлённый генератор.
- Таким образом, значения получаются псевдонезависимы, что достаточно для статистических целей.
- Выбор изначального генератора в этом случае не имеет значения, и можно создать константный генератор с помощью `mkStdGen` — например, `mkStdGen 42`.

Метод Монте-Карло

- Получается, что `replicateM` создаёт *выборку* — список значений случайной величины, причём каждый раз используется обновлённый генератор.
- Таким образом, значения получаются псевдонезависимы, что достаточно для статистических целей.
- Выбор изначального генератора в этом случае не имеет значения, и можно создать константный генератор с помощью `mkStdGen` — например, `mkStdGen 42`.
- Функция `mkStdGen` чистая:

```haskell
mkStdGen :: Int -> StdGen
```

Метод Монте-Карло

- Получается, что `replicateM` создаёт *выборку* — список значений случайной величины, причём каждый раз используется обновлённый генератор.
- Таким образом, значения получаются псевдонезависимы, что достаточно для статистических целей.
- Выбор изначального генератора в этом случае не имеет значения, и можно создать константный генератор с помощью `mkStdGen` — например, `mkStdGen 42`.
- Функция `mkStdGen` чистая:

```
mkStdGen :: Int -> StdGen
```

- Равномерное распределение на $[0, 1)$ даётся функцией `getRandom` с конкретизированным типом:

```
unif = getRandom  :: MonadRandom m => m Double
```

Метод Монте-Карло

```haskell
import Control.Monad.Random

unif = getRandom :: MonadRandom m => m Double

inCircle x y = ( (x-0.5)^2 + (y-0.5)^2 <= 0.25 )

mcCheck = do
    x <- unif
    y <- unif
    return (inCircle x y)

xs = evalRand (replicateM 100000 mcCheck) (mkStdGen 42)
t = 4 * (length [x | x <- xs, x == True])
main = putStrLn $ show t
```

- Эта программа функционально чистая, и потому всегда выдаёт одно и то же значение: $314128 \approx \pi \cdot 100000$.

Метод Монте-Карло

- Эта программа функционально чистая, и потому всегда выдаёт одно и то же значение: $314128 \approx \pi \cdot 100000$.
- Кстати, профилирование показывает, что с ленивостью здесь всё в порядке: вся выборка (100000 булевых значений) в памяти не хранится.

Метод Монте-Карло

- Эта программа функционально чистая, и потому всегда выдаёт одно и то же значение: $314128 \approx \pi \cdot 100000$.
- Кстати, профилирование показывает, что с ленивостью здесь всё в порядке: вся выборка (100000 булевых значений) в памяти не хранится.
- Однако как же сгенерировать «настоящее» случайное число, т.е. взять источник случайности из системы?

Метод Монте-Карло

- Эта программа функционально чистая, и потому всегда выдаёт одно и то же значение: $314128 \approx \pi \cdot 100000$.
- Кстати, профилирование показывает, что с ленивостью здесь всё в порядке: вся выборка (100000 булевых значений) в памяти не хранится.
- Однако как же сгенерировать «настоящее» случайное число, т.е. взять источник случайности из системы?
- Поскольку здесь происходит взаимодействие с «внешним миром», понадобится монада `IO`.

Внешний генератор случайных чисел

- Монада `IO` хранит *глобальный* генератор случайности, который инициализируется в начале работы программы.

Внешний генератор случайных чисел

- Монада `IO` хранит *глобальный* генератор случайности, который инициализируется в начале работы программы.
- Доступ к нему

```
getStdGen :: IO StdGen
```

Внешний генератор случайных чисел

- Монада `IO` хранит *глобальный* генератор случайности, который инициализируется в начале работы программы.
- Доступ к нему

```
getStdGen :: IO StdGen
```

- При этом этот генератор — это просто глобальная переменная, и два вызова `getStdGen` дадут одно и то же.

Внешний генератор случайных чисел

- Монада `IO` хранит *глобальный* генератор случайности, который инициализируется в начале работы программы.
- Доступ к нему

  ```
  getStdGen :: IO StdGen
  ```
- При этом этот генератор — это просто глобальная переменная, и два вызова `getStdGen` дадут одно и то же.
- Генератор можно «обновить» с помощью

  ```
  newStdGen :: IO StdGen
  ```

  но при этом он будет просто заменён на следующее значение псевдослучайной последовательности.

Внешний генератор случайных чисел

- Монада `IO` хранит *глобальный* генератор случайности, который инициализируется в начале работы программы.
- Доступ к нему

```
getStdGen :: IO StdGen
```

- При этом этот генератор — это просто глобальная переменная, и два вызова `getStdGen` дадут одно и то же.
- Генератор можно «обновить» с помощью

```
newStdGen :: IO StdGen
```

  но при этом он будет просто заменён на следующее значение псевдослучайной последовательности.
- Чтобы получить «настоящее» новое случайное число, см. `Data.Random.Source.DevRandom`

Бесконечные вероятностные вычисления

- Ленивость позволяет программировать внутри монады класса `MonadRandom` вычисления со случайными числами, которые при некоторых их значениях длятся бесконечно долго.

Бесконечные вероятностные вычисления

- Ленивость позволяет программировать внутри монады класса `MonadRandom` вычисления со случайными числами, которые при некоторых их значениях длятся бесконечно долго.
- Пример: дана «нечестная» монета, выпадающая орлом с вероятностью p, где $0 < p < 1$, $p \neq 1/2$. Нужно с её помощью сымитировать «честное» бросание, с вероятностью $1/2$.

Бесконечные вероятностные вычисления

- Ленивость позволяет программировать внутри монады класса `MonadRandom` вычисления со случайными числами, которые при некоторых их значениях длятся бесконечно долго.
- Пример: дана «нечестная» монета, выпадающая орлом с вероятностью p, где $0 < p < 1$, $p \neq 1/2$. Нужно с её помощью сымитировать «честное» бросание, с вероятностью $1/2$.
- Соображение: если бросить монету два раза, то последовательности орёл-решка и решка-орёл равновероятны.

Бесконечные вероятностные вычисления

- Ленивость позволяет программировать внутри монады класса `MonadRandom` вычисления со случайными числами, которые при некоторых их значениях длятся бесконечно долго.
- Пример: дана «нечестная» монета, выпадающая орлом с вероятностью p, где $0 < p < 1$, $p \neq 1/2$. Нужно с её помощью сымитировать «честное» бросание, с вероятностью $1/2$.
- Соображение: если бросить монету два раза, то последовательности орёл-решка и решка-орёл равновероятны.
- В случаях орёл-орёл или решка-решка пробуем ещё раз.

Бесконечные вероятностные вычисления

- Ленивость позволяет программировать внутри монады класса `MonadRandom` вычисления со случайными числами, которые при некоторых их значениях длятся бесконечно долго.
- Пример: дана «нечестная» монета, выпадающая орлом с вероятностью p, где $0 < p < 1$, $p \neq 1/2$. Нужно с её помощью сымитировать «честное» бросание, с вероятностью $1/2$.
- Соображение: если бросить монету два раза, то последовательности орёл-решка и решка-орёл равновероятны.
- В случаях орёл-орёл или решка-решка пробуем ещё раз.
- Процесс может оказаться бесконечным, но вероятность этого равна нулю.

Бесконечные вероятностные вычисления

```
unfair = fromList [("heads",0.25),("tails",0.75)]

fair = do
    a <- unfair
    b <- unfair
    if (a /= b) then (return a) else fair
```

Бесконечные вероятностные вычисления

```
unfair = fromList [(”heads”,0.25),(”tails”,0.75)]

fair = do
    a <- unfair
    b <- unfair
    if (a /= b) then (return a) else fair
```

- Если попытаться предвычислить распределение `fair`, то мы уйдём в бесконечный цикл (хотя математически оно равно $(1/2, 1/2)$).

Бесконечные вероятностные вычисления

```
unfair = fromList [("heads",0.25),("tails",0.75)]

fair = do
    a <- unfair
    b <- unfair
    if (a /= b) then (return a) else fair
```

- Если попытаться предвычислить распределение `fair`, то мы уйдём в бесконечный цикл (хотя математически оно равно $(1/2, 1/2)$).
- К счастью, Haskell ленив, и `fair` остаётся как задумка (thunk). Настоящее вычисление произойдёт потом, когда мы уже получим конкретные значения.