

# Функциональное программирование. Вопросы к экзамену

Конспекты

27 января

## Содержание

<b>1. Теория</b> .....	<b>2</b>
1.1. Сравнение функционального и императивного подходов к программированию. Функциональная модель вычислений. ....	2
1.2. Чистое $\lambda$ -исчисление. $\lambda$ -термы, свободные и связанные переменные. Классические комбинаторы, комбинаторная логика. ....	3
1.3. Подстановка, лемма подстановки. Одношаговая и многошаговая $\beta$ -редукция, $\beta$ -эквивалентность. ....	4
1.4. $\alpha$ -эквивалентность. Индексы Де Брауна. $\eta$ -эквивалентность и принцип экстенциональности. ....	5
1.5. Кодирование булевых значений, кортежей в чистом бестиповом $\lambda$ -исчислении. ....	6
1.6. Кодирование чисел Чёрча в чистом бестиповом $\lambda$ -исчислении. Арифметические операции над ними. ....	7
1.7. Теорема о неподвижной точке. $Y$ -комбинатор. Решение рекурсивных уравнений на термы. ....	7
1.8. Нормальная форма. Редукционные графы. ....	7
1.9. Теорема Чёрча-Россера. Параллельная $\beta$ -редукция. Полная эволюция. ....	8
1.10. Следствия теоремы Чёрча-Россера. ....	10
1.11. Стратегии редукции. Теорема о нормализации. Механизмы вызова в функциональных языках. ....	10
1.12. Функция предшествования для чисел Чёрча. Комбинатор примитивной рекурсии. ....	12
1.13. Просто типизированное $\lambda$ -исчисление в стиле Карри. Предтермы. Утверждения о типизации. Контексты. Правила типизации. ....	12
1.14. Просто типизированное $\lambda$ -исчисление в стиле Чёрча. Предтермы. Утверждения о типизации. Контексты. Правила типизации. ....	13
1.15. Свойства систем просто типизированного $\lambda$ -исчисления. ....	14
1.16. Связь между системами Карри и Чёрча. Проблемы разрешимости. Сильная и слабая нормализация. ....	15
1.17. Правило <code>foldr/build</code> и реализация высокоуровневых оптимизаций в GHC. ....	16
1.18. Понятие главного (наиболее общего) типа. Подстановки типа и их композиция. Унификаторы. ....	17
1.19. Алгоритм унификации. ....	18
1.20. Алгоритм построения системы ограничений для типизации терма. ....	18
1.21. Главная пара и главный тип. Теорема Хиндли-Милнера. ....	19
1.22. Обобщения алгоритма Хиндли-Милнера. <code>let</code> -полиморфизм и его ограничения . . .	19
<b>2. Haskell</b> .....	<b>20</b>
2.1. Основы программирования на Haskell. Связывания. Рекурсия. Базовые конструкции языка. Частичное применение. Бесточечный стиль. ....	20
2.2. Основные встроенные типы языка Haskell. Параметрический полиморфизм. Система модулей. ....	20
2.3. Операторы и их сечения в Haskell. ....	20
2.4. Строгие и нестрогие функции. Ленивое и энергичное исполнение. Форсирование, слабая головная нормальная форма. ....	20
2.5. Списки, стандартные функции для работы с ними. Генерация (выделение) списков. ....	20
2.6. Алгебраические типы данных. Сопоставление с образцом, его семантика. Полиморфные и рекурсивные типы данных. ....	20

2.7. Трансляция образцов в Kernel. Синонимы в образцах, ленивые и охранные образцы. Образцы в $\lambda$ - и let-выражениях. ....	21
2.8. Объявления type и newtype. Метки полей. Строгие конструкторы данных. ....	21
2.9. Классы типов. Объявления представителей. Классы типов Eq, Ord, Enum и Bound. ....	21
2.10. Стандартные классы типов: Num и его наследники. ....	22
2.11. Стандартные классы типов: Show и Read. ....	22
2.12. Полугруппы и моноиды. Представители класса типов Monoid. ....	23
2.13. Правая и левая свёртки списков. Энергичные версии. Развертки. ....	23
2.14. Класс типов Foldable и его представители. ....	23
2.15. Класс типов Functor и его представители. ....	24
2.16. Класс типов Applicative и его представители. ....	25
2.17. Классы типов Alternative и MonadPlus и их представители. ....	25
2.18. Аппликативные парсеры. ....	26
2.19. Класс типов Traversable и его представители. ....	27
2.20. Монады. Класс типов Monad. Законы монад. do-нотация. ....	27
2.21. Класс типов MonadFail, его история и представители. ....	27
2.22. Стандартные монады: Maybe и списки. ....	28
2.23. Ввод-вывод в чистых языках. Монада IO. Взаимодействие с файловой системой. ...	28
2.24. Монада Reader. ....	28
2.25. Монада Writer. ....	28
2.26. Монада State. ....	28
2.27. Монада Except. ....	29
2.28. Мультипараметрические классы типов. Роль классов MonadReader, MonadWriter, MonadState и MonadError в mtl. ....	29
2.29. Трансформеры монад. Библиотеки transformers и mtl. ....	30

## 1. Теория

### 1.1. Сравнение функционального и императивного подходов к программированию. Функциональная модель вычислений.

- Императивная модель (Sequential Model of Computation). Программа это последовательность инструкций, изменяющих состояние
  1. Последовательные инструкции
  2. Изменяем состояние
  3. Условия и циклы
  4. Завершается после последней инструкции
- Функциональная модель. Программа это выражение. Ее исполнение это редукция (вычисление) этого выражения
  1. Вычисляются редексы по заданным правилам
  2. Завершается когда не осталось редексов
  3. = это связывание. Рекурсивное связывание

*Примечание:* Что-то про

- Разные стратегии редукции: строгий (правый терм аппликации вычисляется до не редекса), ленивая (самый левый внешний редекс).
- Что-то про каррирование.

## 1.2. Чистое $\lambda$ -исчисление. $\lambda$ -термы, свободные и связанные переменные. Классические комбинаторы, комбинаторная логика.

**Определение 1.2.1:** Множество лямбда термов  $V = \{x, y, z, \dots\}$

$$\begin{aligned}x \in V &\implies x \in \Lambda \\M, N \in \Lambda &\implies (M N) \in \Lambda \\M \in \Lambda, x \in V &\implies (\lambda x. M) \in \Lambda\end{aligned}$$

Или

$$\Lambda ::= V \mid (\Lambda \Lambda) \mid (\lambda V. \Lambda)$$

Примечание: Про приоритеты и ассоциативность

**Определение 1.2.2:**  $\beta$ -редукция

$$(\lambda x. M) N \rightsquigarrow_{\beta} [x \mapsto N]M$$

**Определение 1.2.3:** Множество  $FV(T)$  **свободных** переменных в терме  $T$ :

$$\begin{aligned}FV(x) &= \{x\} \\FV(M N) &= FV(M) \cup FV(N) \\FV(\lambda x. M) &= FV(M) \setminus \{x\}\end{aligned}$$

**Определение 1.2.4:** Множество  $BV(T)$  **связанных** переменных в терме  $T$ :

$$\begin{aligned}BV(x) &= \emptyset \\BV(M N) &= FV(M) \cup FV(N) \\BV(\lambda x. M) &= BV(M) \cup \{x\}\end{aligned}$$

**Определение 1.2.5:**  $(\lambda x. M) N$  —  $\beta$ -редекс

**Определение 1.2.6:**  $M$  — **замкнутый терм** если  $FV(M) = \emptyset$

**Определение 1.2.7:**

$$\begin{aligned}
 I &= \lambda x. x \\
 \omega &= \lambda x. x x \\
 \Omega &= \omega \omega = (\lambda x. x x) (\lambda x. x x) \\
 K &= \lambda x y. x \\
 K_* &= \lambda x y. y \\
 C &= \lambda f x y. f y x \\
 B &= \lambda f g x. f (g x) \\
 S &= \lambda f g x. f x (g x)
 \end{aligned}$$

*Примечание:* Про переименование связанных переменных

**Определение 1.2.8:** Термы  $\alpha$ -эквивалентны если отличаются только именами связанных переменных

*Примечание:* Комбинаторы можно определить как примитив.  $S, K$  – базис

### 1.3. Подстановка, лемма подстановки. Одношаговая и многошаговая $\beta$ -редукция, $\beta$ -эквивалентность.

*Примечание:* Подстановка  $[x \mapsto A]M$  заменяет только свободные  $x$ . Есть соглашение (Барендрегта) что имена связанных всегда выбираем так чтобы они отличались от имен свободных.

**Определение 1.3.1:** Подстановка определена индуктивно:

$$\begin{aligned}
 [x \mapsto N]x &= N \\
 [x \mapsto N]y &= y \\
 [x \mapsto N](P Q) &= ([x \mapsto N]P) ([x \mapsto N]Q) \\
 [x \mapsto N](\lambda x. P) &= \lambda x. P \\
 [x \mapsto N](\lambda y. P) &= \lambda y. [x \mapsto N]P \quad \text{если } y \notin \text{FV}(N) \\
 [x \mapsto N](\lambda y. P) &= \lambda y'. [x \mapsto N]([y \mapsto y']P) \quad \text{если } y \in \text{FV}(N)
 \end{aligned}$$

*Примечание:* Подстановки не коммутируют. Т.е.  $[x \mapsto N]([y \mapsto L]M)$  не обязательно равно  $[y \mapsto L]([x \mapsto N]M)$

**Лемма 1.3.1:** Пусть  $M, N, L \in \Lambda$ . Предположим  $x \neq y$  и  $x \notin \text{FV}(L)$ . Тогда

$$[y \mapsto L]([x \mapsto N]M) \equiv [x \mapsto [y \mapsto L]N]([y \mapsto L]M)$$

*Доказательство:* Индукция по всем случаям ■

**Определение 1.3.2:** Бинарное отношение  $\mathcal{R}$  над  $\Lambda$  **совместимое** если  $\forall M, N, Z \in \Lambda \forall x \in V$

$$\begin{aligned} M \mathcal{R} N &\Rightarrow Z M \mathcal{R} Z N \\ M \mathcal{R} N &\Rightarrow M Z \mathcal{R} N Z \\ M \mathcal{R} N &\Rightarrow \lambda x. M \mathcal{R} \lambda x. N \end{aligned}$$

**Определение 1.3.3:** Наименьшее совместимое отношение  $\rightsquigarrow_\beta$  содержащее правило  $\beta$ :

$$(\lambda x. M) N \rightsquigarrow_\beta [x \mapsto N]M$$

называется **отношением  $\beta$ -редукции**

**Определение 1.3.4:** Бинарное отношение  $\twoheadrightarrow_\beta$  над  $\Lambda$

$$\begin{aligned} M &\twoheadrightarrow_\beta M \text{ (refl)} \\ M \rightsquigarrow_\beta N &\Rightarrow M \twoheadrightarrow_\beta N \text{ (ini)} \\ M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L &\Rightarrow M \twoheadrightarrow_\beta L \text{ (trans)} \end{aligned}$$

Транзитивное рефлексивное замыкание  $\rightsquigarrow_\beta$

Примечание:  $\twoheadrightarrow_\beta^+$  – транзитивное замыкание  $\rightsquigarrow_\beta$

**Определение 1.3.5:** Бинарное отношение  $\beta$ -конвертируемости  $=_\beta$  над  $\Lambda$

$$\begin{aligned} M \twoheadrightarrow_\beta N &\Rightarrow M =_\beta N \text{ (ini)} \\ M =_\beta N &\Rightarrow N =_\beta M \text{ (sym)} \\ M =_\beta N, N =_\beta L &\Rightarrow M =_\beta L \text{ (trans)} \end{aligned}$$

**Утверждение 1.3.1:** Отношение  $\beta$ -конвертируемости является наименьшим отношением эквивалентности, содержащим  $\beta$ -правило

*Доказательство:* Индукция по определениям ■

## 1.4. $\alpha$ -эквивалентность. Индексы Де Брауна. $\eta$ -эквивалентность и принцип экстенциональности.

**Определение 1.4.1:**  $\lambda x. M \rightsquigarrow_\alpha \lambda y. [x \mapsto y]M$ , если  $y \notin \text{FV}(M)$

Примечание: Про De Bruijn.  $\lambda x. (\lambda y. x y) \leftrightarrow \lambda (\lambda 1 0)$

**Определение 1.4.2:** Отношение  $\eta$ -эквивалентности

$$\lambda x. M x \rightsquigarrow_\eta M$$

если  $x \notin \text{FV}(M)$

Примечание: Можно определить совместимое, рефлексивное, симметричное и транзитивное отношение  $\eta$ -эквивалентности. Его смысл в том что вычислительное поведение термов с обеих сторон одинаво:

$$(\lambda x. N x)M =_{\beta} N M$$

Примечание:  $\eta$ -преобразование обеспечивает **принцип экстенциональности**.  $\forall N : F N =_{\beta} G N$

Выбираем  $y \notin \text{FV}(F) \cup \text{FV}(G)$

$$F y =_{\beta} G y$$

$$\lambda y. F y =_{\beta} \lambda y. G y \text{ (правило } \xi)$$

$$F =_{\beta\eta} G$$

## 1.5. Кодирование булевых значений, кортежей в чистом бестиповом $\lambda$ -исчислении.

**Определение 1.5.1:**

$$\mathbf{pair} := \lambda x y f. f x y$$

$$\mathbf{fst} := \lambda p. p \lambda x y. x = \lambda p. p \mathbf{K}$$

$$\mathbf{snd} := \lambda p. p \lambda x y. y = \lambda p. p \mathbf{K}_*$$

Законы пар

$$\forall N M \mathbf{fst} (\mathbf{pair} N M) =_{\beta} N$$

$$\forall N M \mathbf{snd} (\mathbf{pair} N M) =_{\beta} M$$

**Определение 1.5.2:**

$$\mathbf{true} := \lambda x y. x$$

$$\mathbf{false} := \lambda x y. y$$

$$\mathbf{if} := \lambda b x y. b x y =_{\beta\eta} \lambda b. b$$

Законы

$$\forall N M \mathbf{if} \mathbf{true} N M =_{\beta} N$$

$$\forall N M \mathbf{if} \mathbf{false} N M =_{\beta} M$$

## 1.6. Кодирование чисел Чёрча в чистом бестиповом $\lambda$ -исчислении. Арифметические операции над ними.

Определение 1.6.1:

$$\mathbf{0} := \lambda f z. z$$

$$\text{suc} := \lambda n f z. f (n f z)$$

Законы

$$\text{natElim } \bar{n} f z =_{\beta} \underbrace{f (f \dots (f z))}_n \quad \text{Но это не точно}$$

$$\forall f \text{ ini } \text{natElim } \mathbf{0} f \text{ init} =_{\beta} \text{ini}$$

$$\forall n f \text{ ini } \text{natElim} (\text{suc } n) f \text{ ini} =_{\beta} f (\text{natElim } n f \text{ ini})$$

Утверждение 1.6.1: Операции **Доделать**

## 1.7. Теорема о неподвижной точке. $Y$ -комбинатор. Решение рекурсивных уравнений на термы.

**Теорема 1.7.1** (О неподвижной точке): Для любого  $\lambda$ -терма  $F$  существует неподвижная точка.  $\forall F \in \Lambda \exists X \in \Lambda F X =_{\beta} X$ .

*Доказательство:* Введем  $W := \lambda x. F (x x)$  и  $X := W W$ . Тогда

$$X \equiv W W \equiv (\lambda x. F (x x)) W =_{\beta} F (W W) \equiv F X$$

■

**Теорема 1.7.2** (О комбинаторе неподвижной точки): Существует  $Y$ , такой что  $\forall F \in \Lambda F (Y F) =_{\beta} Y F$

*Доказательство:* Введем  $Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$ . Заметим что  $Y F =_{\beta} W W$ . Тогда из предыдущего доказательства:

$$Y F =_{\beta} W W =_{\beta} F (W W) \equiv F (Y F)$$

■

*Примечание:* Как решать рекурсивные уравнения с помощью этого:

$$F N = N F$$

$$F N = (\lambda f n. n f) F N$$

$$F = \underbrace{(\lambda f n. n f) F}_{F'}$$

$$F = Y F'$$

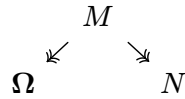
## 1.8. Нормальная форма. Редукционные графы.

**Определение 1.8.1:**  $\lambda$ -терм находится в  $\beta$ -нормальной форме ( $\beta$ -NF) если в нем нет подтермов, являющихся  $\beta$ -редексами

**Определение 1.8.2:**  $\lambda$ -терм имеет  $\beta$ -нормальную форму ( $\beta$ -NF) если для некоторого  $N$  выполняется  $M =_{\beta} N$  и  $N$  находится в  $\beta$ -NF

**Утверждение 1.8.1:** Не все термы имеют  $\beta$ -NF

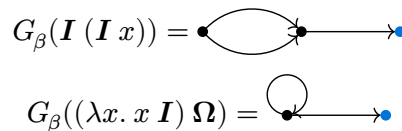
*Пример:*  $\Omega \equiv \omega \omega \rightsquigarrow_{\beta} \omega \omega$  Но это не значит что  $\Omega$  не имеет нормальную форму. Можем существовать какой-то  $M$  и  $N$  в  $\beta$ -NF, такой что



Тогда  $\Omega =_{\beta} N$

**Определение 1.8.3:** **Редукционный граф** терма  $M \in \Lambda$  (обозначаемый  $G_{\beta}(M)$ ) — это ориентированный мультиграф с вершинами в  $\{N \mid M \twoheadrightarrow_{\beta} N\}$  и дугами  $\rightsquigarrow_{\beta}$ .

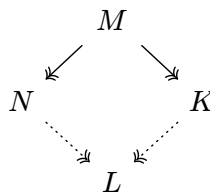
*Пример:*



### 1.9. Теорема Чёрча-Россера. Параллельная $\beta$ -редукция. Полная эволюция.

**Теорема 1.9.1** (Черча-Россера): Если  $M \twoheadrightarrow_{\beta} N$ ,  $M \twoheadrightarrow_{\beta} K$ , то существует  $L$ , такой что  $N \twoheadrightarrow_{\beta} L$  и  $K \twoheadrightarrow_{\beta} L$ .

Примечание:  $\beta$ -редукция обладает **свойством ромба**



Примечание:  $\rightsquigarrow_{\beta}$  не обладает этим свойством

**Определение 1.9.1:** Бинарное отношение **параллельной  $\beta$ -редукции**

- $x \Rightarrow_{\beta} x$  для любой переменной  $x$
- если  $P \Rightarrow_{\beta} P'$ , то  $\lambda x. P \Rightarrow_{\beta} \lambda x. P'$
- если  $P \Rightarrow_{\beta} P'$  и  $Q \Rightarrow_{\beta} Q'$ , то  $P Q \Rightarrow_{\beta} P' Q'$
- если  $P \Rightarrow_{\beta} P'$  и  $Q \Rightarrow_{\beta} Q'$ , то  $(\lambda y. P) Q \Rightarrow_{\beta} [y \mapsto Q'] P'$

Примечание: Это отношение рефлексивно, но не транзитивно — можно сокращать только изначально существовавшие редексы

**Лемма 1.9.1:**

1. Если  $M \rightsquigarrow_{\beta} M'$ , то  $M \Rightarrow_{\beta} M'$
2. Если  $M \Rightarrow_{\beta} M'$ , то  $M \rightarrow_{\beta} M'$
3. Если  $M \Rightarrow_{\beta} M'$  и  $N \Rightarrow_{\beta} N'$ , то  $[x \mapsto N]M \Rightarrow_{\beta} [x \mapsto N']M'$

*Доказательство:*

1. Индукция по определению  $M \rightsquigarrow_{\beta} M'$
2. Индукция по определению  $M \Rightarrow_{\beta} M'$
3. Аналогично (2).

4 случай  $M = (\lambda y. P) Q$ ,  $M' = [y \mapsto Q']P'$ .

ИП:  $[x \mapsto N]P \Rightarrow_{\beta} [x \mapsto N']P'$ ,  $[x \mapsto N]Q \Rightarrow_{\beta} [x \mapsto N']Q'$

$$\begin{aligned}
 [x \mapsto N]M &\equiv [x \mapsto N](\lambda y. P) Q = && \text{определение } \mapsto \\
 &(\lambda y. [x \mapsto N]P) ([x \mapsto N]Q) \Rightarrow_{\beta} && \text{ИП + определение } \Rightarrow_{\beta} \\
 &[y \mapsto [x \mapsto N']Q']([x \mapsto N']P') && \text{Лемма подстановки} \\
 [x \mapsto N']([y \mapsto Q']P') &\equiv [x \mapsto N']M'
 \end{aligned}$$

■

**Определение 1.9.2:** **Полной эволюцией** (complete development) терма  $M$  называют терм  $M^*$ , определяемый индуктивно

$$\begin{aligned}
 x^* &= x \\
 (\lambda x. P)^* &= \lambda x. P^* \\
 (P Q)^* &= P^* Q^* \text{ если } P \text{ не абстракция} \\
 ((\lambda x. P) Q)^* &= [x \mapsto Q^*]P^*
 \end{aligned}$$

Примечание: Отношение  $M \Rightarrow_{\beta} N$  порождается сокращением *некоторых* редексов в  $M$ , а  $M^*$  – сокращением *всех* редексов

*Пример:*  $(\omega (I I))^* = I I$

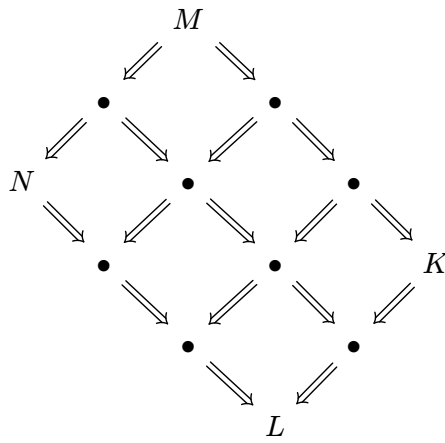
**Лемма 1.9.2** (о полной эволюции): Если  $M \Rightarrow_{\beta} M'$ , то  $M' \Rightarrow_{\beta} M^*$ .

*Доказательство:* индукция по определению  $M \Rightarrow_{\beta} M'$ . ■

Следствие 1.9.2.1: Если  $M \Rightarrow_{\beta} M'$  и  $M \Rightarrow_{\beta} M''$ , то  $M' \Rightarrow_{\beta} M^*$  и  $M'' \Rightarrow_{\beta} M^*$

Примечание: Это свойство ромба  $\Rightarrow_{\beta}$

*Доказательство Черч-Россер:* Если  $M \rightsquigarrow_{\beta} \dots \rightsquigarrow_{\beta} N$  и  $M \rightsquigarrow_{\beta} \dots \rightsquigarrow_{\beta} K$ , то  $M \Rightarrow_{\beta} \dots \Rightarrow_{\beta} N$  и  $M \Rightarrow_{\beta} \dots \Rightarrow_{\beta} K$ . Сцепляя диаграмки



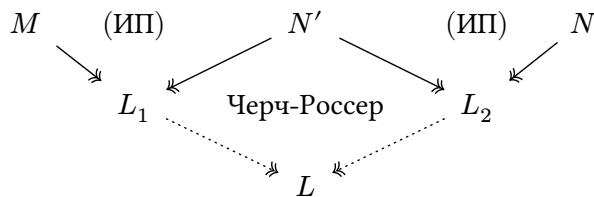
находим  $L$ , такое что  $N \Rightarrow_{\beta} \dots \Rightarrow_{\beta} L$  и  $K \Rightarrow_{\beta} \dots \Rightarrow_{\beta} L$  откуда  $N \twoheadrightarrow_{\beta} \dots \twoheadrightarrow_{\beta} L$  и  $K \twoheadrightarrow_{\beta} \dots \twoheadrightarrow_{\beta} L$  ■

### 1.10. Следствия теоремы Чёрча-Россера.

Следствие 1.9.1.1: Если  $M =_{\beta} N$ , то существует  $L$ , такой что,  $M \twoheadrightarrow_{\beta} L$  и  $N \twoheadrightarrow_{\beta} L$ .

Доказательство: Индукция по генерации  $=_{\beta}$

- $M =_{\beta} N$ , поскольку  $M \twoheadrightarrow_{\beta} N$ . Возьмем  $L \equiv N$
- $M =_{\beta} N$ , поскольку  $N =_{\beta} M$ . По ИП имеется общий  $\beta$ -редукт  $L_1$  для  $N, M$ . Возьмем  $L \equiv L_1$
- $M =_{\beta} N$ , поскольку  $M =_{\beta} N', N' =_{\beta} N$ . Тогда



**Теорема 1.10.1** (Редуцируемость к NF): Если  $M$  имеет  $N$  в качестве  $\beta$ -NF, то  $M \twoheadrightarrow_{\beta} N$

*Примечание:* Теперь можно доказать что у  $\Omega$  нет нормальной формы. Тогда должно было бы выполняться  $\Omega \twoheadrightarrow_{\beta} N$ , но  $\Omega$  редуцируется лишь к себе и не является нормальной формы

**Теорема 1.10.2** (Единственность NF):  $\lambda$ -терм имеет не более одной  $\beta$ -NF

### 1.11. Стратегии редукции. Теорема о нормализации. Механизмы вызова в функциональных языках.

**Определение 1.11.1:** Терм слабо нормализуем ( $WN_{\beta}$ ) если существует последовательность  $\beta$ -редукций приводящих его к  $\beta$ -NF.

**Определение 1.11.2:** Терм сильно нормализуем ( $SN_{\beta}$ ) если любая последовательность  $\beta$ -редукций приводит его в  $\beta$ -NF

**Теорема 1.11.1:**  $\lambda$ -терм можем иметь одну из двух форм.

$$\begin{aligned}\lambda\vec{x}. y \vec{N} &\equiv \lambda x_1 \dots x_n. y N_1 \dots N_k \quad n \geq 0, k \geq 0 \\ \lambda\vec{x}. (\lambda z. M) \vec{N} &\equiv \lambda x_1 \dots x_n. (\lambda z. M) N_1 \dots N_k \quad n \geq 0, k > 0\end{aligned}$$

**Определение 1.11.3:** Первая форма называется **головная нормальная форма (HNF)**: переменная  $y$  называется **головной переменной**, а редекс  $(\lambda z. M) N_1$  — **головным редексом**. Переменная  $y$  можем совпадать с одной из  $x_i$

**Определение 1.11.4:** **Слабая головная нормальная форма (WHNF)** — это HNF или лямбда-абстракция, то есть не редекс на верхнем уровне.

**Определение 1.11.5:** Стратегия редукции  $F$  — это отображение множества  $\Lambda$  в себя, тождественное для нормальных форм и обладающее свойством  $M \rightsquigarrow_\beta F(M)$  для прочих термов.

**Определение 1.11.6:** Нормальная (крайне левая) стратегия  $F_1$ :

$$\begin{aligned}F_1(\lambda\vec{x}. y \vec{P} Q \vec{R}) &= \lambda\vec{x}. y \vec{P} F_1(Q) \vec{R} \quad \text{если } \vec{P} \in \text{NF}_\beta \text{ и } Q \notin \text{NF}_\beta \\ F_1(\lambda\vec{x}. (\lambda z. M) Q \vec{R}) &= \lambda\vec{x}. [z \mapsto Q]M \vec{R}\end{aligned}$$

**Определение 1.11.7:** Аппликативная стратегия стратегия  $F_a$ :

$$\begin{aligned}F_a(\lambda\vec{x}. y \vec{P} Q \vec{R}) &= \lambda\vec{x}. y \vec{P} F_a(Q) \vec{R} \quad \text{если } \vec{P} \in \text{NF}_\beta \text{ и } Q \notin \text{NF}_\beta \\ F_a(\lambda\vec{x}. (\lambda z. M) Q \vec{R}) &= \lambda\vec{x}. [z \mapsto Q]M \vec{R} \quad \text{если } Q \in \text{NF}_\beta \\ F_a(\lambda\vec{x}. (\lambda z. M) Q \vec{R}) &= \lambda\vec{x}. (\lambda z. M) F_a(Q) \vec{R} \quad \text{если } Q \notin \text{NF}_\beta\end{aligned}$$

**Определение 1.11.8:** Стратегия редукции называется *нормализующей* если для любого  $M \in \text{WN}_\beta$  существует конечное  $i \in \mathbb{N}$  такое что  $F^i(M) \in \text{NF}_\beta$

**Утверждение 1.11.1:** Аппликативная стратегий не нормализующая

**Пример:**  $K I \Omega$ . У него есть нормальная форма  $I$ . Однако аппликативная стратегия “зависнет” на вычислении  $\Omega$ .

**Теорема 1.11.2 (о нормализации):** Нормальная стратегия  $F_1$  является нормализующей

Примечание: Теперь можем доказать отсутствие  $NF_\beta$  у терма. Например,  $K \Omega I$ .

Попробуем прорецидуировать нормальной стратегией:

$$K \Omega I \rightarrow_\beta \Omega \rightsquigarrow_\beta \Omega \rightsquigarrow_\beta \dots$$

Понятно что не существует  $i \in \mathbb{N}$  такого что  $F_1(K \Omega I)$  будет в нормальной форме. А т.к.  $F_1$  нормализуема, то получается что у терма нет нормальной формы вообще.

**Но это не точно**

Примечание:

- Нормальная стратегия может быть неэффективна: один и тот же терм придется считать несколько раз
- Нормальная стратегия ленивая

## 1.12. Функция предшествования для чисел Чёрча. Комбинатор примитивной рекурсии.

Примечание: Цикл по интервалу:

```
def rec(f, ini, n):
    state = ini
    for i in range(0, n):
        state = f(i, state)
    return state
```

**Определение 1.12.1:**

$$\text{rec } f \text{ ini } n =_\beta f(n-1) (f(n-2) (\dots (f 1 (f 0 \text{ ini})) \dots))$$

Как это сделать:

```
start ini := pair 0 ini
step f (pair i res) := pair (suc i) (f i res)
iter f ini n := natElim n (step f) (start ini)
rec f ini n := snd (iter f ini n)
```

Примечание:  $\text{pred } n := \text{rec } K 0 n$

## 1.13. Просто типизированное $\lambda$ -исчисление в стиле Карри. Предтермы. Утверждения о типизации. Контексты. Правила типизации.

Примечание: Термы те же что и в бестиповом. Каждый терм обладает множеством различных типов.

**Определение 1.13.1:** Множество типов  $\mathbb{T}$  системы  $\lambda_{\rightarrow}$  определяется индуктивно

$$\begin{aligned} \alpha, \beta, \dots \in \mathbb{T} & \quad (\text{переменные типа}) \\ A, B \in \mathbb{T} \implies (A \rightarrow B) \in \mathbb{T} & \quad (\text{типы пространства функций}) \end{aligned}$$

или

$$\mathbb{T} ::= \mathbb{V} \mid (\mathbb{T} \rightarrow \mathbb{T})$$

где  $\mathbb{V} = \{\alpha, \beta, \dots\}$

**Определение 1.13.2:** Множество **предтермов** (или **псевдотермов**)  $\Lambda$  строится из переменных из  $V = \{x, y, z, \dots\}$  с помощью аппликации и абстракции

$$\begin{aligned} x \in V &\Rightarrow x \in \Lambda \\ M, N \in \Lambda &\Rightarrow (M N) \in \Lambda \\ M \in \Lambda, x \in V &\Rightarrow (\lambda x. M) \in \Lambda \end{aligned}$$

или

$$\Lambda ::= V \mid (\Lambda \Lambda) \mid (\lambda V. \Lambda)$$

Примечание: Предтермы это в точности термы  $\lambda$

**Определение 1.13.3:** Утверждение типизации в  $\lambda_{\rightarrow}$  “а ля Карри” имеет вид

$$M : A$$

где  $M \in \Lambda$  и  $A \in \mathbb{T}$ . Тип  $A$  иногда называют **предикатом**, а терм  $M$  — **субъектом** утверждения

**Определение 1.13.4:** **Объявление** — утверждение типизации с термовой переменной в качестве субъекта

**Определение 1.13.5:** **Контекст** — это множество объявлений с различными переменными в качестве субъекта

$$\Gamma = \{x_1^{A_1}, \dots, x_n^{A_n}\}$$

Примечание: Можно рассматривать как частичную функцию  $V \rightarrow \mathbb{T}$

**Определение 1.13.6:** Утверждение  $M : C$  называется **выводимым** в контексте  $\Gamma$

$$\Gamma \vdash M : C$$

если его вывод может быть произведен по правилам

$$\begin{aligned} \text{(аксиома)} \quad & \Gamma \vdash x : A \quad \text{если } x^A \in \Gamma \\ (\rightarrow E) \quad & \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} \\ (\rightarrow I) \quad & \frac{\Gamma, x^A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B} \end{aligned}$$

Если для предтерма  $M$  существуют  $\Gamma$  и  $C$  такие что  $\Gamma \vdash M : C$ , то его называют (**допустимым**) **термом**

## 1.14. Просто типизированное $\lambda$ -исчисление в стиле Чёрча. Предтермы. Утверждения о типизации. Контексты. Правила типизации.

*Примечание:* Термы – аннотированные версии бестиповых термов. Каждый терм имеет тип, выводимый из способа, которым терм аннотирован

**Определение 1.14.1:** Множество **предтермов**  $\Lambda_{\mathbb{T}}$  строится из переменных из  $V = \{x, y, z, \dots\}$  с помощью аппликации и аннотированной типами абстракции:

$$\Lambda_{\mathbb{T}} ::= V \mid (\Lambda_{\mathbb{T}} \Lambda_{\mathbb{T}}) \mid (\lambda V^{\mathbb{T}}. \Lambda_{\mathbb{T}})$$

*Примечание:* Утверждение типизации см. [Определение 1.13.3](#). Только вместо  $\Lambda$  –  $\Lambda_{\mathbb{T}}$

*Примечание:* Правила см. [Определение 1.13.6](#). Только в абстракции  $x$  аннотирован

## 1.15. Свойства систем просто типизированного $\lambda$ -исчисления.

**Лемма 1.15.1** (об инверсии (генерации)):

- $\Gamma \vdash x : A \Rightarrow x^A \in \Gamma$
- $\Gamma \vdash M N : B \Rightarrow \exists A \Gamma \vdash M : A \rightarrow B \wedge \Gamma \vdash N : A$
- $\Gamma \vdash \lambda x. M : C \Rightarrow \exists A, B \Gamma, x^A \vdash M : B \wedge C \equiv A \rightarrow B$
- $\Gamma \vdash \lambda x^A. M : C \Rightarrow \exists B \Gamma, x^A \vdash M : B \wedge C \equiv A \rightarrow B$

**Лемма 1.15.2** (о типизируемости подтерма): Пусть  $M'$  – подтерм  $M$ . Тогда  $\Gamma \vdash M : A \Rightarrow \Gamma' \vdash M' : A'$  для некоторых  $\Gamma'$  и  $A'$ .

**Лемма 1.15.3** (разбавления (thinning)): Пусть  $\Gamma, \Delta$  – контексты, причем  $\Delta \supseteq \Gamma$ . Тогда  $\Gamma \vdash M : A \Rightarrow \Delta \vdash M : A$ . Расширение контекста не влияет на выводимость утверждения типизации.

**Лемма 1.15.4** (о свободных переменных):  $\Gamma \vdash M : A \Rightarrow \text{FV}(M) \subseteq \text{dom}(\Gamma)$ . Свободные переменные типизированного терма должны присутствовать в контексте

**Лемма 1.15.5** (сужения):  $\Gamma \vdash M : A \Rightarrow \Gamma \upharpoonright \text{FV}(M) \vdash M : A$ . Сужение контекста до множества свободных переменных терма не влияет на выводимость утверждения типизации.

**Свойство:** Рассмотрим предтерм  $x x$ . Предположим что это терм. Тогда имеются  $\Gamma$  и  $B$ , такие что

$$\Gamma \vdash x x : B$$

По лемме об инверсии существует такой  $A$ , что правый подтерм  $x : A$ , а левый подтерм имеет тип  $A \rightarrow B$ . По лемме о контекстах  $x \in \text{dom}(\Gamma)$  и должен иметь там единственное связывание по определению контекста. То есть  $A \equiv A \rightarrow B$  – тип является подвыражением себя, чего не может быть, поскольку типы конечны. По лемме о типизируемости подтерма предтермы  $\omega, \Omega, Y$  не имеют типа

**Определение 1.15.1:** Для типов  $A, B \in \mathbb{T}$  подстановку  $A$  вместо переменной типа  $\alpha$  в  $B$  обозначим  $[\alpha \mapsto A]B$ .

**Лемма 1.15.6** (подстановки типа):

- $\lambda_{\rightarrow}$  Карри.  $\Gamma \vdash M : B \Rightarrow [\alpha \rightarrow A]\Gamma \vdash M : [\alpha \mapsto A]B$
- $\lambda_{\rightarrow}$  Черч.  $\Gamma \vdash M : B \Rightarrow [\alpha \rightarrow A]\Gamma \vdash [\alpha \mapsto A]M : [\alpha \mapsto A]B$

**Лемма 1.15.7** (подстановки терма): Пусть  $\Gamma, x^A \vdash M : B$  и  $\Gamma \vdash N : A$ , тогда  $\Gamma \vdash [x \mapsto N]M : B$

**Теорема 1.15.1** (о редукции субъекта): Пусть  $M \rightarrow_{\beta} N$ . Тогда  $\Gamma \vdash M : A \Rightarrow \Gamma \vdash N : A$ .

Следствие 1.15.1.1: Множество типизируемых в  $\lambda_{\rightarrow}$  термов замкнуто относительно редукции

**Теорема 1.15.2** (о единственности типа  $\lambda_{\rightarrow}$  а ля Черч): Пусть  $\Gamma \vdash_C M : A$  и  $\Gamma \vdash_C M : B$ . Тогда  $A \equiv B$

Следствие 1.15.2.1: Пусть  $\Gamma \vdash_C M : A, \Gamma \vdash_C N : B$  и  $M =_{\beta} N$ . Тогда  $A \equiv B$ .

## 1.16. Связь между системами Карри и Чёрча. Проблемы разрешимости. Сильная и слабая нормализация.

Зададим на термах стирающее отображение  $|\cdot| : \Lambda_{\mathbb{T}} \rightarrow \Lambda$

$$\begin{aligned} |x| &= x \\ |M N| &= |M| |N| \\ |\lambda x^A. M| &= \lambda x. |M| \end{aligned}$$

Все атрибутированные типами термы из версии Черча  $\lambda_{\rightarrow}$  “проектируются” в термы в версии Карри.

$$M \in \Lambda_{\mathbb{T}} \wedge \Gamma \vdash_C M : A \Rightarrow \Gamma \vdash |M| : A$$

Термы из версии Карри  $\lambda_{\rightarrow}$  могут быть “подняты” в термы из версии Черча

$$M \in \Lambda \wedge \Gamma \vdash_C M : A \Rightarrow \exists N \in \Lambda_{\mathbb{T}} \Gamma \vdash_C N : A \wedge |N| \equiv M$$

Для произвольного типа  $A \in \mathbb{T}$  выполняется

$$A \text{ обитаем в } \lambda_{\rightarrow} \text{ Карри} \iff A \text{ обитаем в } \lambda_{\rightarrow} \text{ Черч}$$

Примечание: Проблемы:

- $\vdash M : A?$  Задача проверки типа (Type Checking)
- $\vdash M : ?$  Задача синтеза типа (Type Synthesis)
- $\vdash ? : A$  Задача обитаемости типа (Type Inhabitation)

**Определение 1.16.1:** Систему типов называют **слабо нормализуемой**, если все ее допустимые термы слабо нормализуемы

**Определение 1.16.2:** Систему типов называют **сильно нормализуемой**, если все ее допустимые термы сильно нормализуемы

**Теорема 1.16.1:** Обе системы  $\lambda_{\rightarrow}$  сильно нормализуемы

## 1.17. Правило `foldr/build` и реализация высокоуровневых оптимизаций в GHC.

Операция противоположная `fold` это

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr g init = todo
```

Другой способ сделать развертку

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

Делаем списки на функциях

```
> g c n = c 'H' (c 'i' c '!' n)
> build g
"Hi!"
```

Если отсутствуют вызовы `seq`, то имеем место

$$\text{foldr } f \ z \ (\text{build } g) \equiv g \ f \ z$$

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr c n [] = n
foldr c n (x : xs) = x `c` (foldr c n xs)
```

Можем выражать высокоуровневые оптимизации:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

iterateFB :: (a -> b -> b) -> (a -> a) -> a -> b
iterateFB c f y = go y where go x = x `c` go (f x)
```

```
{-# RULES
"iterate"    [-1] forall f x. iterate f x = build (\c _ -> iterateFB c f x)
"fold/build" forall k z g. foldr k z (build g) = g k z
"iterateFB"  [1] iterateFB (:) = iterate
#-}
{-# INLINE [1] build #-}
```

- Сначала заменяем вызовы `iterate` на вызов `build`
- Если были вызовы формата `foldr k z (iterate f x)`. То после предыдущего пункта они стали `foldr k z (build (\c _ -> iterateFB c f x))`. Вторым правило заменим вызов `foldr` на прямой вызов нашей лямбды: `(\c _ -> iterateFB c f x) k z`.
- Заменим оставшиеся вызовы `iterateFB` для построения списка обратно на `iterate`.

Таким образом избавились от построения промежуточных списков. Но это не точно

## 1.18. Понятие главного (наиболее общего) типа. Подстановки типа и их композиция. Унификаторы.

**Определение 1.18.1:** В версии Карри терму можно приписать множество типов. Тип называется **главным** (**principal**) если из него можно получить любой другой через подстановку.

**Определение 1.18.2:** Подстановка типа это операция  $S : \mathbb{T} \rightarrow \mathbb{T}$  такая что

$$S(A \rightarrow B) \equiv S(A) \rightarrow S(B)$$

*Примечание:* Обычно подстановка тождественна на всех переменных кроме конечного носителя  $\text{sup}(S) = \{\alpha \mid S(\alpha) \neq \alpha\}$ .

**Лемма 1.18.1** (подстановки): Для  $\alpha_1 \neq \alpha_2$  и  $\alpha_1 \notin \text{FV}(A_2)$  верно равенство

$$[\alpha_2 := A_2]([\alpha_1 := A_1]B) \equiv [\alpha_1 := [\alpha_2 := A_2]A_1]([\alpha_2 := A_2]B)$$

**Определение 1.18.3:** **Композиция двух подстановок** — подстановка с носителем, являющимся объединением их носителей, над которым последовательно выполнены обе подстановки

*Пример:*

- $S = [\alpha := \gamma \rightarrow \beta, \beta := \alpha \rightarrow \gamma]$
- $T = [\alpha := \beta \rightarrow \gamma, \gamma := \beta]$

$$\begin{aligned} T \circ S &= [\alpha := T(S(\alpha)), \beta := T(S(\beta)), \gamma := T(S(\gamma))] \\ &= [\alpha := \beta \rightarrow \beta, \beta := (\beta \rightarrow \gamma) \rightarrow \beta, \gamma := \beta] \end{aligned}$$

**Утверждение 1.18.1:** Подстановки образуют моноид относительно  $\circ$  с  $[\ ]$  в роли нейтрального элемента

**Лемма 1.18.2** (о композиции подстановок): Если подстановка определена как композиция элементарных

$$[\alpha_1 := A'_1, \dots, \alpha_n := A'_n] \equiv [\alpha_n := A_n] \circ \dots \circ [\alpha_1 := A_1]$$

причем  $\forall i \leq j \alpha_i \notin \text{FV}(A_j)$ , то

$$\forall i, j \alpha_i \notin \text{FV}(A'_j)$$

*Доказательство:* Индукция с использованием Леммы подстановки ■

**Определение 1.18.4:** **Унификатор** для типов  $A$  и  $B$  это подстановка  $S$ , такая что  $S(A) \equiv S(B)$ .

**Определение 1.18.5:**  $S$  это **главный унификатор** для  $A$  и  $B$ , если для любого другого унификатора  $S'$  существует подстановка  $T$ , такая что  $S' \equiv T \circ S$ .

### 1.19. Алгоритм унификации.

**Теорема 1.19.1:** Существует алгоритм унификации  $U$ , который для заданных типов  $A$  и  $B$  возвращает:

- Главный унификатор  $S$  для  $A$  и  $B$ , если  $A$  и  $B$  могут быть унифицированы
- Сообщение об ошибке в противном случае

Алгоритм унификации  $U$ :

$$\begin{aligned} U(\alpha, \alpha) &= [] \\ U(\alpha, B) \mid \alpha \in \text{FV}(B) &= \text{ошибка} \\ U(\alpha, B) \mid \alpha \notin \text{FV}(B) &= [\alpha := B] \\ U(A_1 \rightarrow A_2, \alpha) &= U(\alpha, A_1 \rightarrow A_2) \\ U(A_1 \rightarrow A_2, B_1 \rightarrow B_2) &= U(U_2 A_1, U_2 B_1) \circ U_2, \text{ где } U_2 = U(A_2, B_2) \end{aligned}$$

### 1.20. Алгоритм построения системы ограничений для типизации терма.

Пример:

$$\lambda x^\alpha y^\beta. y^\beta \left( \underbrace{\lambda z^\gamma. y^\beta x^\alpha}_{\varepsilon} \right)^\delta$$

1. Припишем типовую (мета-)переменную всем термовым переменным  $x^\alpha, y^\beta$
2. Припишем типовую (мета-)переменную всем аппликативным подтермам  $(y x) : \delta$
3. Выпишем уравнения (ограничения) на типы, необходимые для типизируемости терма:  $\beta \sim \alpha \rightarrow \delta, \beta \sim (\gamma \rightarrow \delta) \rightarrow \varepsilon$
4. Найдем главный унификатор для типовых переменных, дающий решения уравнений:

$$\alpha := \gamma \rightarrow \delta, \beta := (\gamma \rightarrow \delta) \rightarrow \varepsilon, \delta := \varepsilon$$

Главный тип  $\lambda x y. y (\lambda z. y x) : (\gamma \rightarrow \varepsilon) \rightarrow ((\gamma \rightarrow \varepsilon) \rightarrow \varepsilon) \rightarrow \varepsilon$

**Обозначение:**  $S$ , унифицирующее систему уравнений на типы  $E = \{A_1 \sim B_1, \dots, A_n \sim B_n\}$  введем обозначение  $S \vDash E$ .

**Теорема 1.20.1:** Для любых терма  $M \in \Lambda$ , контекста  $\Gamma$  ( $\text{FV}(M) \subseteq \text{dom}(\Gamma)$ ) и типа  $A \in \mathbb{T}$  существует конечное множество уравнений на типы  $E = E(\Gamma, M, A)$ , такое что для любой подстановки  $S$

- $S \vDash E(\Gamma, M, A) \Rightarrow S(\Gamma) \vdash M : S(A)$
- $S(\Gamma) \vdash M : S(A) \Rightarrow S' \vDash E(\Gamma, M, A)$  для некоторой  $S'$ , имеющего тот же эффект, что и  $S$ , на типовых переменных в  $A$  и  $\Gamma$

Алгоритм построения системы ограничений  $E$ :

$$\begin{aligned}
 E(\Gamma, x, A) &= \{A \sim \Gamma(x)\} \\
 E(\Gamma, P Q, A) &= E(\Gamma, P, \alpha \rightarrow A) \cup E(\Gamma, Q, \alpha) \\
 E(\Gamma, \lambda x. P, A) &= E((\Gamma, x^\alpha), P, \beta) \cup \{\alpha \rightarrow \beta \sim A\}
 \end{aligned}$$

где переменные  $\alpha, \beta$  – свежие

### 1.21. Главная пара и главный тип. Теорема Хиндли-Милнера.

**Определение 1.21.1:** Для  $M \in \Lambda$  **главной парой** называют пару  $(\Gamma, A)$  такую что

- $\Gamma \vdash M : A$
- $\Gamma' \vdash M : A' \Rightarrow \exists S S(\Gamma) \subseteq \Gamma' \wedge S(A) \equiv A'$

**Теорема 1.21.1 (Хиндли-Милнера):** Существует алгоритм PP, возвращающий для  $M \in \Lambda$

- главную пару  $(\Gamma, A)$ , если  $M$  имеет тип
- сообщение об ошибке в противном случае

Пусть  $FV(M) = \{x_1, \dots, x_n\}$ . Выберем произвольные различные переменные  $\alpha_0, \dots, \alpha_n$  и сконструируем  $\Gamma_0 = \{x_1^{\alpha_1}, \dots, x_n^{\alpha_n}\}$

Алгоритм PP

$$\begin{aligned}
 PP(M) \mid U(E(\Gamma_0, M, \alpha_0)) &\equiv \text{ошибка} = \text{ошибка} \\
 PP(M) \mid U(E(\Gamma_0, M, \alpha_0)) &\equiv S = (S(\Gamma_0), S(\alpha_0))
 \end{aligned}$$

**Определение 1.21.2:** Для  $M \in \Lambda^0$  **главным типом** называют тип  $A$ , такой что

- $\vdash M : A$
- $\vdash M : A' \Rightarrow \exists S S(A) \equiv A'$

**Следствие 1.21.1.1:** Существует алгоритм PT, возвращающий для  $M \in \Lambda^0$

- главный тип  $A$ , если  $M$  имеет тип
- сообщение об ошибке в противном случае

*Примечание:*  $\Lambda^0$  – комбинаторы Но это не точно

### 1.22. Обобщения алгоритма Хиндли-Милнера. let-полиморфизм и его ограничения

Можно расширить ХМ из:

- листья образованы только переменными
- узлы образованы только  $(->) :: * -> * -> *$

в:

- листья – конструкторы типа кайнда  $*$
- узлы – конструкторы типа любых стрелочных кайндов
- узлы – полиморфные, например,  $m$  а или  $t$  m а

ХМ не справляется на  $\backslash f -> (f 'z', f \text{ True})$ . Конструкция `let ... in ...` не просто сахар для лямбды, это отдельный примитив. Можем писать `let f = \x -> x in (f 'z', f True)`. В таком случае `f :: forall a. a -> a`.

Однако не все можно написать: `let f g = (g 'c', g True) in f id`. `let` может только в поверхностные кванторы, а в примере выше хотим тип  $(\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \langle \text{Char}, \text{Bool} \rangle$ . Нужно включить `RankNTypes` и явно указать тип `f`.

## 2. Haskell

### 2.1. Основы программирования на Haskell. Связывания. Рекурсия.

**Базовые конструкции языка. Частичное применение.**

**Бесточечный стиль.**

Очев. + guards

*Примечание:* Разверт туплов от 2 до 15 (по стандарту), 63 в GHC

### 2.2. Основные встроенные типы языка Haskell. Параметрический полиморфизм. Система модулей.

`Bool, Char, Int, Integer, Float, Double, -, []`

### 2.3. Операторы и их сечения в Haskell.

`(2 ***) ≡ (***) 2 ≡ \y -> 2 *** y`

`(*** 3) ≡ \x -> x *** 3`

### 2.4. Строгие и нестрогие функции. Ленивое и энергичное исполнение.

**Форсирование, слабая головная нормальная форма.**

*Примечание:*

`ignore x = r2`

`ignore undefined` не упадет, потому что `ignore` **не строгая** по аргументу `x`.

`seq :: a -> b -> b`

`seq ⊥ b = ⊥`

`seq a b = b`

Это не настоящий код. Но фактически `seq` вычисляет первый аргумент чтобы проверить что это не `⊥`. Вычисляет оно до WHNF — лямбды или конструктора

`infixr 0 $!`

`($!) :: (a -> b) -> a -> b`

`f $! x = x `seq` f x`

*Примечание:* Неэффективный факториал

`factorial n = helper 1 n where`

`helper acc k | k > 1 = helper (acc * k) (k - 1)`  
`| otherwise = acc`

В `acc` будет собираться `thunk (... ((1 * n) * (n - 1)) * (n - 2) * ... * 2)` потому что `helper` не строгий по `acc`. Можем форсировать вычисление:

`factorial n = helper 1 n where`

`helper acc k | k > 1 = (helper $! (acc * k)) (k - 1)`  
`| otherwise = acc`

### 2.5. Списки, стандартные функции для работы с ними. Генерация (выделение) списков.

*Примечание:* Выделение списка (List Comprehension)

`[ [x, y] | x <- "ABC", y <- "de" ] = ["Ad", "Ae", "Bd", ...]`

### 2.6. Алгебраические типы данных. Сопоставление с образцом, его семантика. Полиморфные и рекурсивные типы данных.

*Примечание:* Паттерн матчинг делает функция строгой по этому аргументу

## 2.7. Трансляция образцов в Kernel. Синонимы в образцах, ленивые и охранные образцы. Образцы в $\lambda$ - и let-выражениях.

Примечание: Ленивые паттерны

```
(**) :: (a -> c) -> (b -> d) -> (a, b) -> (c, d)
(**) f g ~(x, y) = (f x, g y)
```

Всегда успешно, связывание откладывается до момента использования

Использование паттернов в let также транслируется в ленивые паттерны

```
let p = e1 in e  $\equiv$  case e1 of ~p -> e
```

Примечание: Более мощные гарды

```
firstOdd :: [Integer] -> Integer
firstOdd xs | Just x <- find odd xs = x
            | otherwise = 0

firstOddIsBig :: [Integer] -> Bool
firstOddIsBig xs
  | Just x <- find odd xs, x > 1000 = True
  | otherwise
```

## 2.8. Объявления type и newtype. Метки полей. Строгие конструкторы данных.

Примечание: Можно конструировать указывая не все поля, ошибка вылетит если попытаться использовать неинициализированные.

Примечание: Так можно

```
data Homo = Known {name :: String, male :: Bool}
          | Unknown {male :: Bool}
```

Примечание: Можно форсировать вычисление “полей”

```
infix 6 :+
data Complex a = !a :+ !a
```

Но

```
GHCI> case 1 :+ undefined of _ -> 42
42
```

Т.к. конструктор не был вычислен

## 2.9. Классы типов. Объявления представителей. Классы типов Eq, Ord, Enum и Bounded.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
  {-# MINIMAL (==) | (/=) #-}

class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

Законы Ord:

```
 $\forall x. x \leq x$  -- Reflexivity
 $\forall x y z. x \leq y \ \&\& \ y \leq z \equiv x \leq z$  -- Transitivity
 $\forall x y. x \leq y \ \&\& \ y \leq x \equiv x == y$  -- Antisymmetry
 $\forall x. x \leq y \ || \ y \leq x$  -- Comparability
```

Примечание: Про OrphanInstances

Примечание: Про GeneralizedNewtypeDeriving

Примечание: Про DerivingStrategies

```
{-# LANGUAGE DerivingStrategies #-}
newtype Temperature = Temperature {getTemp :: Double}
  deriving (Num,Eq)
  deriving newtype Show
```

Примечание: Про фантомные типовые параметры

Минимальное полное определение toEnum, fromEnum

```
class Enum a where
  succ, pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a] -- [n..]
  enumFromThen :: a -> a -> [a] -- [n,n'..]
  enumFromTo :: a -> a -> [a] -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n,n'..m]
```

```
class Bounded a where
  minBound, maxBound :: a
```

## 2.10. Стандартные классы типов: Num и его наследники.

MCD: все, кроме negate или (-)

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

```
x - y = x + negate y
negate x = 0 - x
```

Подклассы:

- Integral — целочисленное деление (через Real) (Integer, Int)
- Fractional — обычное деление (Float, Double)

Примечание: Преобразования

- fromIntegral :: (Num b, Integral a) => a -> b
- ceiling, floor, truncate, round :: (RealFrac a, Integral b) => a -> b

Примечание: Рациональные дроби

```
data Ratio a = !a :% !a deriving (Eq)
(%) :: Integral a => a -> a -> Ratio a
numerator, denominator :: Integral a => Ratio a -> a
```

```
type Rational = Ratio Integer
```

Преобразование в рациональные числа:

- toRational float
- approxRational 4.9 0.1 = 49 % 10

## 2.11. Стандартные классы типов: Show и Read.

```
type ShowS = String -> String
class Show a where
  show :: a -> String
```

```
shows :: a -> ShowS
```

Примечание: С помощью shows можем гарантированно складывать строки начиная с конца, это будет линейно. Не собираем промежуточные строки

```
type ReadS a = String -> [(a, String)]
```

```
class Read a where
  read :: String -> a

  reads :: ReadS a
```

## 2.12. Полугруппы и моноиды. Представители класса типов Monoid.

MCD: (<>), sconcat

```
infixr 6 <>
class Semigroup a where
  (<>) :: a -> a -> a

  sconcat :: NonEmpty a -> a

  stimes :: Integral b => b -> a -> a
  stimes = stimesDefault
```

Законы

$$(x \langle \rangle y) \langle \rangle z \equiv x \langle \rangle (y \langle \rangle z)$$

Пример: Список

```
class Semigroup a => Monoid a where
  {-# MINIMAL mempty | mconcat #-}
  mempty :: a
  mempty = mconcat []

  -- In a future GHC release will be removed
  mappend :: a -> a -> a
  mappend = (<>)

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Законы

$$\begin{aligned} \text{mempty} \langle \rangle x &\equiv x \\ x \langle \rangle \text{mempty} &\equiv x \\ \text{mconcat} &\equiv \text{foldr } (\langle \rangle) \text{ mempty} \end{aligned}$$

Пример: Список, для Bool All, Any, для чисел Sum, Product, Min

## 2.13. Правая и левая свёртки списков. Энергичные версии. Развертки.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = x `f` (foldr f z xs)
```

Примечание: У левой свертки foldl таскается большой thunk, поэтому есть строгая версия foldl'

Примечание: foldr1, foldl1, scanl

Примечание:

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr g ini
  | Nothing <- next = []
  | Just (a,b) <- next = a : unfoldr g b
  where next = g ini
```

## 2.14. Класс типов Foldable и его представители.

```
class Foldable t where
  foldr, foldr' :: (a -> b -> b) -> b -> t a -> b
  foldr f z t = appEndo (foldMap (Endo . f) t) z

  foldl, foldl' :: (a -> b -> a) -> a -> t b -> a
```

```
foldl f z t = appEndo
  (getDual (foldMap (Dual . Endo . flip f) t)) z
```

```
foldr1, foldl1 :: (a -> a -> a) -> t a -> a
```

```
fold :: Monoid m => t m -> m
fold = foldMap id
```

```
foldMap :: Monoid m => (a -> m) -> t a -> m
foldMap f = foldr (mappend . f) mempty
```

```
{-# MINIMAL foldMap | foldr #-}
```

```
toList :: t a -> [a]
```

```
null :: t a -> Bool
null = foldr (\_ _ -> False) True
```

```
length :: t a -> Int
length = foldl' (\n _ -> n + 1) 0
```

```
elem :: Eq a => a -> t a -> Bool
```

```
sum, product :: Num a => t a -> a
sum = getSum . foldMap Sum
maximum, minimum :: Ord a => t a -> a
```

*Пример:* Список, Maybe, Either, туплы

Функции обобщенные до использования Foldable

```
concat :: Foldable t => t [a] -> [a]
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]
```

```
and,or :: Foldable t => t Bool -> Bool
any,all :: Foldable t => (a -> Bool) -> t a -> Bool
```

```
maximumBy,minimumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a
```

```
notElem :: (Foldable t, Eq a) => a -> t a -> Bool
```

```
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
```

## 2.15. Класс типов Functor и его представители.

```
infixl 4 <$, <$>, $>
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  (<$) = fmap . const
```

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
($>) :: Functor f => f a -> b -> f b
($>) = flip (<$)
```

*Пример:* Список, Maybe, какой-нибудь Tree, Either, тупл, (->) e

Другие полезные функции

```
void :: Functor f => f a -> f ()
void xs = () <$ xs
```

```
infixl 1 <&>
(<&>) :: Functor f => f a -> (a -> b) -> f b
xs <&> g = g <$ xs
```

```
unzip :: Functor f => f (a, b) -> (f a, f b)
unzip xs = (fst <$> xs, snd <$> xs)
```

Законы

```
fmap id ≡ id
fmap (f . g) ≡ fmap f . fmap g
```

*Примечание:* Для чего нельзя правильно написать функтор:

```
newtype Endo a = Endo { appEndo :: a -> a }
```

А для перевернутой стрелки вообще нельзя написать функтор

## 2.16. Класс типов `Applicative` и его представители.

```
infixl 4 <*>, *>, <*, <*>
class Functor f => Applicative f where
  {-# MINIMAL pure, ((<*>) | liftA2) #-}
  pure :: a -> f a

  (<*>) :: f (a -> b) -> f a -> f b
  (<*>) = liftA2 id

  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  liftA2 g a b = g <$> a <*> b

  (*>) :: f a -> f b -> f b
  u *> v = (id <$ u) <*> v

  (<*) :: f a -> f b -> f a
  u <*> v = liftA2 const u v
```

*Пример:* Maybe, список: каждый с каждым, попарно ZipList, пара с Monoid

Законы

```
fmap g . pure ≡ pure . g
fmap g xs ≡ pure g <*> xs
pure id <*> v ≡ v
pure g <*> pure x ≡ pure (g x)
u <*> pure x ≡ pure ($ x) <*> u
pure (.) <*> u <*> v <*> x ≡ u <*> (v <*> x)
```

## 2.17. Классы типов `Alternative` и `MonadPlus` и их представители.

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
infixl 3 <|>
```

*Примечание:* Население

- Maybe — первый не Nothing (как обретка First)
- ZipList — если первый кончился, он дополняется элементами из второго

Для IO

```
instance Alternative IO where
  empty :: IO a
  empty = failIO "mzero"

  (<|>) :: IO a -> IO a -> IO a
  m <|> n = m `catchException` \( _ :: IOError) -> n
class (Alternative m, Monad m) => MonadPlus m where
  mzero :: m a
  mzero = empty
```

```
mplus :: m a -> m a -> m a
mplus = (<|>)
```

Законы

```
mzero >>= k ≡ mzero
v >> mzero ≡ mzero
```

И как минимум одного из двух:

```
(a `mplus` b) >>= k ≡ (a >>= k) `mplus` (b >>= k)
return a `mplus` b ≡ return a
```

*Примечание:* guard

```
asum :: (Foldable t, Alternative f) => t (f a) -> f a
asum = foldr (<|>) empty
```

```
msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
msum = asum -- foldr mplus mzero
```

```
mfilter :: MonadPlus m => (a -> Bool) -> m a -> m a
mfilter p ma = do
  a <- ma
  if p a
  then return a
  else mzero
```

## 2.18. Аппликативные парсеры.

```
newtype Parser tok a =
  Parser { runParser :: [tok] -> Maybe ([tok],a) }
```

```
satisfy :: (tok -> Bool) -> Parser tok tok
satisfy pr = Parser f where
  f (c:cs) | pr c = Just (cs,c)
  f _ = Nothing
```

```
lower :: Parser Char Char
lower = satisfy isLower
```

```
char :: Char -> Parser Char Char
char c = satisfy (== c)
```

```
instance Functor (Parser tok) where
  fmap :: (a -> b) -> Parser tok a -> Parser tok b
  fmap g = Parser . (fmap . fmap . fmap) g . runParser
```

```
digit :: Parser Char Int
digit = digitToInt <$> satisfy isDigit
```

```
instance Applicative (Parser tok) where
  pure :: a -> Parser tok a
  pure x = Parser $ \s -> Just (s, x)
  (<*>) :: Parser tok (a -> b) -> Parser tok a -> Parser tok b
  Parser u <*> Parser v = Parser f
  where
    f xs = case u xs of
      Nothing -> Nothing
      Just (xs', g) -> case v xs' of
        Nothing -> Nothing
        Just (xs'', x) -> Just (xs'', g x)
```

```
instance Alternative (Parser tok) where
  empty :: Parser tok a
  empty = Parser $ \_ -> Nothing
```

```
(<|>) :: Parser tok a -> Parser tok a -> Parser tok a
Parser u <|> Parser v = Parser f
  where
    f xs = case u xs of
      Nothing -> v xs
      z -> z
```

## 2.19. Класс типов Traversable и его представители.

MCP: sequenceA или traverse

```
class (Functor t, Foldable t) => Traversable t where
  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequenceA = traverse id

  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse g = sequenceA . fmap g
```

Законы

```
traverse Identity ≡ Identity
traverse (Compose . fmap f . g) ≡ Compose . fmap (traverse f) . traverse g
  h . traverse f ≡ traverse (h . f)
```

Эти законы гарантируют

- Траверсы не пропускают элементов
- Траверсы посещают элементы не более одного раза
- traverse pure дает pure
- Траверсы не изменяют исходную структуру — она либо сохраняется, либо полностью исчезает

## 2.20. Монады. Класс типов Monad. Законы монад. do-нотация.

```
infixl 1 >>, >>=
class Applicative m => Monad m where
  {-# MINIMAL (>>=) #-}
  (>>=) :: m a -> (a -> m b) -> m b -- произносят bind

  (>>) :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2

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

*Примечание:* Экземпляры: Maybe

Законы

```
return a >>= k ≡ k a
m >>= return ≡ m
(m >>= k) >>= k' ≡ m >>= (\x -> k x >>= k')
```

Законы в терминах join

```
join . return ≡ id
join . fmap return ≡ id
join . fmap join ≡ join . join
```

## 2.21. Класс типов MonadFail, его история и представители.

```
class Monad m => MonadFail m where
  fail :: String -> m a

instance MonadFail Maybe where
  fail _ = Nothing
```

Закон: fail s это левый ноль для >>=

```
fail s >>= k ≡ fail s
```

## 2.22. Стандартные монады: Maybe и списки.

*Примечание:* Списки flat'ятся

## 2.23. Ввод-вывод в чистых языках. Монада IO. Взаимодействие с файловой системой.

```
newtype IO a =
  IO (State# RealWorld -> (# State# RealWorld, a #))

getChar :: IO Char
getLine :: IO String
getContents :: IO String

putChar :: Char -> IO ()
putStr, putStrLn :: String -> IO ()
print :: Show a => a -> IO ()

interact :: (String -> String) -> IO ()
```

## 2.24. Монада Reader.

```
instance Monad (Reader r) where
  return x = reader $ \_ -> x
  m >>= k = reader $ \e -> let v = runReader m e
                          in runReader (k v) e

reader :: (r -> a) -> Reader r a
runReader :: Reader r a -> r -> a
```

*Примечание:* ask, asks, local

## 2.25. Монада Writer.

```
instance Monoid w => Monad (Writer w) where
  return x = writer (x, mempty)

  m >>= k = let (x,u) = runWriter m
              (y,v) = runWriter $ k x
              in writer (y, u `mappend` v)

writer :: (a, w) -> Writer w a
runWriter :: Writer w a -> (a, w)
```

*Примечание:* tell, listen :: Writer w a -> Writer w (a, w), listens, censor :: (w -> w) -> Writer w a -> Writer w a

## 2.26. Монада State.

```
newtype State s a = State { runState :: s -> (a,s) }

state :: (s -> (a,s)) -> State s a
runState :: State s a -> s -> (a,s)

instance Monad (State s) where
  return x = state $ \st -> (x,st)
  m >>= k = state $
    \st -> let (x,st') = runState m st
            m' = k x
            in runState m' st'

get :: State s s
get = state $ \s -> (s,s)

put :: s -> State s ()
put s = state $ \_ -> ((),s)

modify :: (s -> s) -> State s ()
modify f = do s <- get
```

```
put (f s)
```

```
gets :: (s -> a) -> State s a
gets f = do s <- get
        return (f s)
```

Примечание: replicateM

Примечание:

- Монада ST, runST :: (forall s. ST s a) -> a. Можно делать “нечистые” вычисления в чистых функциях
- IORef это STRef без локальности и соответствующих гарантий безопасности
- MVar это IORef с блокировками
- TVar это изменяемые ячейка памяти в рамках STM

## 2.27. Монада Except.

```
newtype Except e a = Except { runExcept :: Either e a }
```

```
except :: Either e a -> Except e a
except = Except
```

```
instance Monad (Except e) where
  return :: a -> Except e a
  return a = except $ Right a
```

```
(>=>) :: Except e a -> (a -> Except e b) -> Except e b
m >=> k = case runExcept m of
  Left e -> except $ Left e
  Right x -> k x
```

```
throwE :: e -> Except e a
throwE = except . Left
```

```
catchE :: Except e a -> (e -> Except e' a) -> Except e' a
m `catchE` h = case runExcept m of
  Left l -> h l
  Right r -> except $ Right r
```

## 2.28. Мультипараметрические классы типов. Роль классов MonadReader, MonadWriter, MonadState и MonadError в mtl.

```
class Mult a b c where
  (**) :: a -> b -> c

instance Mult Matrix Matrix Matrix where
  {- ... -}
instance Mult Matrix Vector Vector where
  {- ... -}
instance Mult Matrix Int Matrix where
  {- ... -}
instance Mult Int Matrix Matrix where
  {- ... -}
```

Но не можем сделать `Matrix (Vector 1 2) (Vector 3 4) ** Matrix (Vector 1 0) (Vector 0 1)`, т.к. параметр `c` нигде явно не указан и нельзя определить инстанс `Mult`.

Можно определить “функциональную зависимость”. Т.е. для каких-то `a`, `b` не может двух инстансов для двух разные `c` и `c'`

```
class Mult a b c | a b -> c where
  (**) :: a -> b -> c
```

Это используется в mtl. Можно наделить любую монаду свойствами одной из других:

```

class Monad m => MonadReader r m | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a

class (Monoid w, Monad m) => MonadWriter w m | m -> w where
  tell :: w -> m ()
  listen :: m a -> m (a, w)

class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()

class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a

```

## 2.29. Трансформеры монад. Библиотеки transformers и mtl.

**Определение 2.29.1:** Трансформер монад — конструктор типа, который принимает монаду в качестве аргумента и возвращает монаду как результат

*Примечание:*

```

class MonadTrans t where
  lift :: Monad m => m a -> t m a

```

*Пример:*

```

newtype MyMonadT m a = MyMonadT { runMyMonadT :: m (MyMonad a) }

```

*Примечание:* Про fail

Законы

$$\begin{aligned} \text{lift} \ . \ \text{return} &\equiv \text{return} \\ \text{lift} \ (m \gg= k) &\equiv \text{lift} \ m \gg= (\text{lift} \ . \ k) \end{aligned}$$