Функциональное программирование
Лекция 1. Лямбда-исчисление

Денис Николаевич Москвин

СПбГУ, факультет МКН,
бакалавриат «Современное программирование», 2 курс

04.09.2025

План лекции

1. Функциональная модель вычислений

2. Чистое λ-исчисление

3. Подстановка и отношения редукции

План лекции

Императивное программирование

Вычисление описывается в терминах *инструкций*, изменяющих *состояние* вычислителя.

Императивный вычислитель:

- исполняет инструкции *последовательно* `C1; C2; C3`;
- изменяет состояние, следуя инструкциям *присваивания* значений *изменяемым переменным*: `v = value`;
- понимает механизмы *условного исполнения* и *циклов*: инструкции `if, switch, for, while`;
- завершает вычисление, когда достигается последняя инструкция.

Такую модель вычислений иногда называют Sequential Model of Computation.

```
x = 2 * x + 3;
```

Функциональное программирование

Программа это *выражение*, её выполнение — вычисление (*редукция*) этого выражения.

Функциональный вычислитель:

- находит в выражении *редексы*, то есть подвыражения, которые могут быть вычислены непосредственно;
- выполняет сокращения редексов по заданным *правилам вычислений*, обычно выраженным в терминах *подстановки*;
- завершает вычисление, когда редексов не остается.

Пусть вычислитель умеет складывать и умножать.

$$2 * 7 + 3 \quad \leadsto_{*} \quad 14 + 3 \quad \leadsto_{+} \quad 17$$

- На каждом шаге редекс всего один.
- В процессе вычисления могут возникать новые редексы.

Связывание

Символ равенства (=) в ФП задает не присваивание, а *связывание*.

```
z = 2 * 7 + 3
```

- Имя слева связывается с выражением справа, порождая пользовательское вычислительное правило.
- Этим правилом можно пользоваться при вычислениях наряду со встроенными, z в выражениях теперь является редексом

$$
\begin{array}{l}
\texttt{z * 4 + 1} \quad \rightsquigarrow_{z} \\
\texttt{(2 * 7 + 3) * 4 + 1} \quad \rightsquigarrow_{*} \\
\texttt{(14 + 3) * 4 + 1} \rightsquigarrow \quad \texttt{...} \rightsquigarrow \texttt{69}
\end{array}
$$

- Иногда связывание называют *равенством-по-определению* (definitional equality).

Связывание и рекурсия

- Пример с первого слайда допустим и в функциональных языках

```
x = 2 * x + 3
```

- Но его смысл совершенно другой, это *рекурсивное* связывание.
- Вычисления по этому правилу *расходятся*, поскольку не содержат терминирующего условия:

$$
\begin{array}{l}
x * 5 \rightsquigarrow_x \\
(2 * x + 3) * 5 \rightsquigarrow_x \\
(2 * (2 * x + 3) + 3) * 5 \rightsquigarrow_x \\
(2 * (2 * (2 * x + 3) + 3) + 3) * 5 \rightsquigarrow_x \ldots
\end{array}
$$

- Рекурсивное связывание — мощный, но опасный инструмент.

Определение функций

Выражение `2 * y + 3` не содержит редексов, если для `y` нет вычислительного правила. Но при подстановке конкретного числа вместо абстрактного `y` редекс появится.
Следующее выражение представляет собой *анонимную функцию* или *лямбда-абстракцию*:

```
\y -> 2 * y + 3
```

Справа от стрелки (`->`) находится *тело*, а между лямбдой (`\`) и стрелкой — *абстрактор*.
Синтаксис *применения* функции к фактическому аргументу

```
(\y -> 2 * y + 3) 7

(\y -> 2 * y + 3) (4 + 6)
```

Скобки используют исключительно для группировки.

Вычисление: β-редукция

Выражение, в котором анонимная функция применяется к фактическому аргументу, называется β-*редексом*. Вычислительное правило, заключающееся в подстановке в тело лямбда-абстракции фактического значения аргумента вместо формального называется β-*редукцией*.

```
(\y -> 2 * y + 3) 7   ⇝β   2 * 7 + 3   ⇝*   14 + 3   ⇝+   17
```

Связывание позволяет закодировать *именованную* функцию

```
foo = \y -> 2 * y + 3
```

Пример использования

```
foo 7   ⇝foo (\y -> 2 * y + 3) 7   ⇝β   ...  ⇝   17
```

Стратегии редукции

```
(\y -> 2 * y + 3) (4 + 6)   ⇝ ???
```

Ой. А тут два редекса. Какой сокращать?

Стратегии редукции

```
(\y -> 2 * y + 3) (4 + 6)   ⇝ ???
```

Ой. А тут два редекса. Какой сокращать?
В Haskell используют *ленивую стратегию*, сокращая самый левый внешний редекс:

```
(\y -> 2 * y + 3) (4 + 6)   ⇝β  2 * (4 + 6) + 3   ⇝+
2 * 10 + 3   ⇝*  20 + 3   ⇝+  23
```

В языках семейства ML используют *энергичную стратегию*:

```
(\y -> 2 * y + 3) (4 + 6)   ⇝+  (\y -> 2 * y + 3) 10   ⇝β
2 * 10 + 3   ⇝*  20 + 3   ⇝+  23
```

- Редукция — не *функция* над термами, а *отношение*.
- Здесь результаты вычислений одинаковы, но всегда ли это будет так?

Пример рекурсивной функции

Теперь можем рекурсивно определить не константу, а функцию

```
fact = \n -> if n == 0 then 1 else n * fact (n - 1)
```

Пример ленивого вычисления выражения `fact 3`

$$
\begin{array}{l}
\texttt{fact 3} \ \leadsto_{\texttt{fact}} \\
\texttt{(\textbackslash n -> if n == 0 then 1 else n * fact (n-1)) 3} \ \leadsto_{\beta} \\
\texttt{if 3 == 0 then 1 else 3 * fact (3-1)} \ \leadsto_{==} \\
\texttt{if False then 1 else 3 * fact (3-1)} \ \leadsto_{\texttt{if}} \\
\texttt{3 * fact (3-1)} \ \leadsto_{\texttt{fact}} \\
\texttt{3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)} \ \leadsto_{\_} \\
\texttt{...} \\
\texttt{3 * (2 * (1 * if (((3-1)-1)-1) == 0 then 1 else ...))} \ \leadsto_{\_} \\
\texttt{...} \\
\texttt{3 * (2 * (1 * 1))} \ \leadsto_{*} \ \texttt{...} \ \leadsto_{*} \ \texttt{6}
\end{array}
$$

Реализация может быть эффективнее, но «подстановочная» семантика должна сохраняться.

Функции нескольких переменных

Выражение `2 * m + 3 * n` содержит две переменные. Абстракция по одной из них вносит асимметрию

```
\n -> 2 * m + 3 * n
```

Продолжая абстракцию, получим *замкнутое* выражение

```
\m -> (\n -> 2 * m + 3 * n)
```

Вычисление требует последовательной передачи аргументов

```
((\m -> (\n -> 2 * m + 3 * n)) 15) 4   ⇝β
(\n -> 2 * 15 + 3 * n) 4   ⇝β
2 * 15 + 3 * 4  ⇝   ...   ⇝   42
```

Такие функции называют *каррированными*.
Внутренний редекс иллюстрирует понятие *частичного применения* каррированной функции.

Сильные стороны ФП

- Регулярный и лаконичный синтаксис.
- Мощная типизация, необременительная благодаря эффективным алгоритмам вывода типов.
- Возможность легко доказывать свойства программ алгебраическими методами.
- Возможность генерации программ по набору свойств.
- Высокоуровневые оптимизации на базе эквивалентных преобразований.

План лекции

1. Функциональная модель вычислений
2. Чистое λ-исчисление
3. Подстановка и отношения редукции

Определение

Множество λ-***термов*** Λ индуктивно строится из переменных $V = \{x, y, z, \ldots\}$ с помощью *применения* и *абстракции*:

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

- В абстрактном синтаксисе

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

- **Соглашение.** Произвольные термы пишем заглавными буквами, переменные — строчными.

Примеры термов

Примеры λ-термов

$$x$$
$$(x\,z)$$
$$(\lambda x.\,(x\,z))$$
$$((\lambda x.\,(x\,z))\,y)$$
$$(\lambda y.\,((\lambda x.\,(x\,z))\,y))$$
$$((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w)$$
$$(\lambda z.\,(\lambda w.\,((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w)))$$

В этом списке каждый следующий терм содержит предыдущие в качестве *подтермов*.

Соглашения о скобках в термах

Общеприняты следующие соглашения:

- Внешние скобки опускаются
- Применение ассоциативно *влево*:

$$F\,X\,Y\,Z \quad \text{обозначает} \quad (((F\,X)\,Y)\,Z)$$

- Абстракция ассоциативна *вправо*:

$$\lambda x\,y\,z.\,M \quad \text{обозначает} \quad (\lambda x.\,(\lambda y.\,(\lambda z.\,(M))))$$

- Тело абстракции простирается вправо насколько это возможно:

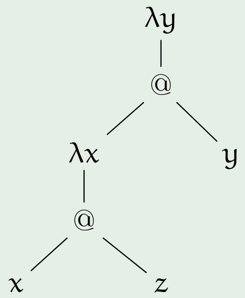
$$\lambda x.\,F\,X\,Y \quad \text{обозначает} \quad \lambda x.\,(F\,X\,Y)$$

Те же примеры, что и выше, но с использованием соглашений

$$
\begin{aligned}
x &= x \\
(x\,z) &= x\,z \\
(\lambda x.\,(x\,z)) &= \lambda x.\,x\,z \\
((\lambda x.\,(x\,z))\,y) &= (\lambda x.\,x\,z)\,y \\
(\lambda y.\,((\lambda x.\,(x\,z))\,y)) &= \lambda y.\,(\lambda x\,.\,x\,z)\,y \\
((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w) &= (\lambda y.\,(\lambda x\,.\,x\,z)\,y)\,w \\
(\lambda z.\,(\lambda w.\,((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w))) &= \lambda z\,w.\,(\lambda y.\,(\lambda x\,.\,x\,z)\,y)\,w
\end{aligned}
$$

Альтернативный способ представления терма — дерево.

Терм $\lambda y.\,(\lambda x\,.\,x\,z)\,y$ в виде AST



Скобки важны лишь при линейном строковом представлении термов.

Редукция как правило вычислений, неформально

- Вводят вычислительное правило β-*редукции*

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

где $[x \mapsto N]\, M$ обозначает *подстановку* N вместо x в M.
- Применение вида $(\lambda x.\, M)\, N$, в которой левый аппликанд является абстракцией, называют β-*редексом*.
- Шаг вычисления по приведенному выше правилу называют *сокращением* редекса.
- В *чистом* λ-исчислении нет ничего кроме переменных, применения, абстракции и β-редукции.
- Точные определения подстановки и *отношения* β-редукции мы дадим чуть позже.

Абстракция $\lambda x\,.\,M[x]$ *связывает* дотоле *свободную* переменную x в терме M.

$$(\lambda y.\,(\lambda w.\,w\,z)\,y)\,x$$

Переменные y и w — связанные, а z и x — свободные.

Связывание переменной ограничивает ее область видимости телом лямбды. Поэтому вне тела лямбды то же самое имя может связываться повторно.

$$(\lambda {\color{red}x}.\,(\lambda {\color{green}x}.\,{\color{green}x}\,z)\,{\color{red}x})\,{\color{blue}x}$$

Переменная x — связанная (дважды!) и свободная, а z — свободная.

Свободные и связанные переменные, определения

Определение

Множество $\mathrm{FV}(\mathrm{T})$ ***свободных (free) переменных*** в терме T:

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

Определение

Множество $\mathrm{BV}(\mathrm{T})$ ***связанных (bound) переменных*** в терме T:

$$
\begin{aligned}
\mathrm{BV}(x) &= \varnothing; \\
\mathrm{BV}(\mathrm{M\,N}) &= \mathrm{BV}(\mathrm{M}) \cup \mathrm{BV}(\mathrm{N}); \\
\mathrm{BV}(\lambda x.\,\mathrm{M}) &= \mathrm{BV}(\mathrm{M}) \cup \{x\}.
\end{aligned}
$$

Определение

M — ***замкнутый*** λ***-терм*** (или ***комбинатор***), если $FV(M) = \varnothing$. Множество замкнутых λ-термов обозначается Λ^0.

Многие комбинаторы имеют общепринятые имена:

Классические комбинаторы

$$\begin{array}{lcl}
\mathbf{I} & = & \lambda x.\, x \\
\boldsymbol{\omega} & = & \lambda x.\, x\, x \\
\boldsymbol{\Omega} & = & \boldsymbol{\omega}\, \boldsymbol{\omega} \;=\; (\lambda x.\, x\, x)(\lambda x.\, x\, x) \\
\mathbf{K} & = & \lambda x\, y.\, x \\
\mathbf{K}_* & = & \lambda x\, y.\, y \\
\mathbf{C} & = & \lambda f\, x\, y.\, f\, y\, x \\
\mathbf{B} & = & \lambda f\, g\, x.\, f\, (g\, x) \\
\mathbf{S} & = & \lambda f\, g\, x.\, f\, x\, (g\, x)
\end{array}$$

Переименование связанных переменных

Имена связанных переменных не важны; их можно (почти) безболезненно переименовывать

$$\begin{array}{lcl} \mathbf{I} & = & \lambda x.\, x \;=\; \lambda y.\, y \;=\; \lambda z.\, z \\ \mathbf{B} & = & \lambda f\, g\, x.\, f\,(g\, x) = \lambda u\, v\, z.\, u\,(v\, z) \end{array}$$

Это верно в операционном смысле: термы с переименованиями при вычислениях дают один и тот же результат

$$\begin{array}{lcl} (\lambda x.\, x)\, N & \rightsquigarrow_\beta & N \\ (\lambda y.\, y)\, N & \rightsquigarrow_\beta & N \\ (\lambda z.\, z)\, N & \rightsquigarrow_\beta & N \end{array}$$

Термы, отличающиеся только именами связанных переменных, называют α-эквивалентными.

Комбина́торная логика

- Комбинаторы можно определить как примитивы, задав их вычислительное поведение на аргументах-переменных

Классические комбинаторы

$$\begin{array}{lcl} \mathbf{I}\,x & = & x \\ \boldsymbol{\omega}\,x & = & x\,x \\ \mathbf{K}\,x\,y & = & x \\ \mathbf{S}\,f\,g\,x & = & f\,x\,(g\,x) \end{array}$$

- Вычисление опять задается подстановкой определения с заменой формальных аргументов на фактические

$$\boldsymbol{\omega}\,\mathbf{I} \rightsquigarrow \mathbf{I}\,\mathbf{I} \rightsquigarrow \mathbf{I}.$$

- Можно выбрать конечный базис. Достаточно $\mathbf{S}$ и $\mathbf{K}$, чтобы получить исчисление, эквивалентное λ-исчислению.

План лекции

Подстановка: нюансы и проблемы

Подстановка выполняется только вместо свободных вхождений:

$$[x \mapsto \lambda z.\, z](x\ (\lambda x.\, x\, y)\ x) = (\lambda z.\, z)\ (\lambda x.\, x\, y)\ (\lambda z.\, z)$$

Проблема *захвата переменной* (variable capture):

$$[x \mapsto y](\lambda y.\, x\, y) = \lambda y.\, y\, y$$

Соглашение Барендрегта

Имена связанных переменных всегда будем выбирать так, чтобы они отличались от имён свободных переменных.

Тогда коллизий, связанных с захватом, можно избежать:

$$[x \mapsto y](\lambda y'.\, x\, y') = \lambda y'.\, y\, y'$$

Однако удобнее встроить переименование в определение подстановки.

Определение подстановки $[x \mapsto N]\, M$

Подстановка терма N вместо свободных вхождений переменной x в терм M задается индукцией по структуре M:

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

Подразумевается, что x отлично от y, а y' — свежая, то есть $y' \notin FV(N) \cup FV(P)$.

Лемма подстановки

Подстановки не коммутируют. Однако верна

Лемма подстановки

Пусть $M, N, L \in \Lambda$. Предположим $x \not\equiv y$ и $x \notin FV(L)$. Тогда

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

Доказательство

Нудная индукция по всем 6 случаям, с разбором всех подслучаев. ∎

Отношение β-редукции за один шаг

Бинарное отношение $\mathcal{R}$ над Λ называют ***совместимым***, если для любых $M, N, Z \in \Lambda$ и любой переменной x

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

Наименьшее совместимое отношение $\rightsquigarrow_\beta$, содержащее

$$(\lambda x.\, M)N \rightsquigarrow [x \mapsto N]\, M \qquad (\text{правило } \beta)$$

называется ***отношением*** β***-редукции***.

$$\underline{(\lambda x\, y.\, x)\,(\lambda a.\, a)}\,(\lambda b.\, b) \rightsquigarrow_\beta \underline{(\lambda y\, a.\, a)\,(\lambda b.\, b)} \rightsquigarrow_\beta \lambda a.\, a$$

Многошаговая β-редукция

Определение

Бинарное отношение $\twoheadrightarrow_\beta$ над Λ:

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

$\twoheadrightarrow_\beta$ является транзитивным рефлексивным замыканием $\rightsquigarrow_\beta$.

Примеры

$$\begin{array}{lcl} (\lambda x y.\, x)\,(\lambda a.\, a)\,(\lambda b.\, b) & \twoheadrightarrow_\beta & (\lambda x y.\, x)\,(\lambda a.\, a)\,(\lambda b.\, b) \\ (\lambda x y.\, x)\,(\lambda a.\, a)\,(\lambda b.\, b) & \twoheadrightarrow_\beta & (\lambda y\, a.\, a)\,(\lambda b.\, b) \\ (\lambda x y.\, x)\,(\lambda a.\, a)\,(\lambda b.\, b) & \twoheadrightarrow_\beta & \lambda a.\, a \end{array}$$

Вводят еще и $\twoheadrightarrow_\beta^+$ — транзитивное замыкание $\rightsquigarrow_\beta$.

Отношение конвертируемости $=_\beta$

Определение

Бинарное отношение $=_\beta$ над Λ:

$$\begin{array}{rcll} M \twoheadrightarrow_\beta N & \Rightarrow & M =_\beta N & \quad (\mathrm{ini}) \\ M =_\beta N & \Rightarrow & N =_\beta M & \quad (\mathrm{sym}) \\ M =_\beta N, N =_\beta L & \Rightarrow & M =_\beta L & \quad (\mathrm{trans}) \end{array}$$

Утверждение

Отношение β-конвертируемости является наименьшим отношением эквивалентности, содержащим β-правило.

Доказательство

Индукция по определениям. ■

Отношение конвертируемости $=_\beta$ (2)

Интуитивно: два терма M и N связаны отношением $=_\beta$, если есть связывающая их цепочка $\rightsquigarrow_\beta$ -стрелок:

$$\begin{tikzcd}
M \arrow[dr, squiggly] & & & & & & & \\
& \bullet \arrow[dr, squiggly] & & \bullet \arrow[dl, squiggly] \arrow[dr, squiggly] & & & & N \arrow[dl, squiggly] \\
& & \bullet & & \bullet \arrow[dr, squiggly] & & \bullet \arrow[dl, squiggly] & \\
& & & & & \bullet & &
\end{tikzcd}$$

Пример. $\mathbf{K}\ \mathbf{I}\ =_\beta\ \mathbf{I}\ \mathbf{I}\ \mathbf{K}_*$

$$\begin{tikzcd}
& & & \mathbf{I}\,\mathbf{I}\,\mathbf{K}_* \arrow[dl, squiggly] \\
\mathbf{K}\ \mathbf{I} \arrow[dr, squiggly] & & \mathbf{I}\ \mathbf{K}_* \arrow[dl, squiggly] & \\
& \mathbf{K}_* & &
\end{tikzcd}$$

Отношение α-эквивалентности

Можно формально определить α-эквивалентность на основе

$$\lambda x.\, M \ \rightsquigarrow_\alpha \ \lambda y.\, [x \mapsto y]\, M, \text{ если } y \notin FV(M) \quad (\text{правило } \alpha).$$

Для рассуждений достаточно соглашения Барендрегта, но для компьютерной реализации α-преобразование полезно:

Пусть $\boldsymbol{\omega} = \lambda x.\, x\, x$ и $\mathbf{1} = \lambda y\, z.\, y\, z$. Тогда

$$\begin{array}{rcl}
\boldsymbol{\omega}\,\mathbf{1} & = & (\lambda x.\, x\, x)\,(\lambda y\, z.\, y\, z) \\
 & =_\beta & (\lambda y\, z.\, y\, z)\,(\lambda y\, z.\, y\, z) \\
 & =_\beta & \lambda z.\, (\lambda y\, z.\, y\, z)\, z \\
 & =_\alpha & \lambda z.\, (\lambda y\, z'.\, y\, z')\, z \\
 & =_\beta & \lambda z\, z'.\, z\, z' \\
 & =_\alpha & \lambda y\, z'.\, y\, z' \\
 & =_\alpha & \lambda y\, z.\, y\, z \ = \ \mathbf{1}
\end{array}$$

- ***Индексы Де Брауна (De Bruijn)*** представляют альтернативный способ представления термов.
- Связанные переменные не именуются, а индексируются, индекс показывает, сколько лямбд «назад» переменная была связана:

$$\lambda x.\,(\lambda y.\,x\,y) \quad \leftrightarrow \quad \lambda\,(\lambda\,1\,0)$$
$$\lambda x.\,x\,(\lambda y.\,x\,y\,y) \quad \leftrightarrow \quad \lambda\,0\,(\lambda\,1\,0\,0)$$

- Свободные переменные при этом получают индексы, превышающие число лямбд слева:

$$\lambda x.\,z\,x\,y \quad \leftrightarrow \quad \lambda\,2\,0\,1$$

- При таком представлении все α-эквивалентные термы кодируются одинаково, и коллизий захвата переменной не возникает.

Отношение η-эквивалентности

- Рассмотрим правило

$$\lambda x.\, M\, x \;\rightsquigarrow_\eta\; M \qquad \text{(правило } \eta\text{)}$$

в предположении, что $x \notin FV(M)$.

- Можно формально определить η-эквивалентность на основе этого правила, добавив совместимость, а также рефлексивность, симметричность и транзитивность.
- Смысл η-эквивалентности в том, что вычислительное поведение термов слева и справа от знака $=_\eta$ одинаково; для произвольного N верно

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

Функциональная экстенсиональность

- η-преобразование обеспечивает ***принцип экстенсиональности***: две функции считаются экстенсионально эквивалентными, если они дают одинаковый результат при любом одинаковом вводе:

$$\forall N \ : \ F\,N =_\beta G\,N.$$

- Выбирая $y \notin FV(F) \cup FV(G)$, получаем (правило ξ, затем правило η)

$$\begin{aligned} F\,y &=_\beta G\,y \\ \lambda y.\,F\,y &=_\beta \lambda y.\,G\,y \\ F &=_{\beta\eta} G \end{aligned}$$

- В Haskell так называемый «бесточечный» стиль записи основан на η-преобразовании.