Функциональное программирование

Лекция 3

Степан Львович Кузнецов

НИУ ВШЭ, факультет компьютерных наук

Типы в языках программирования

- В большинстве языков программирования имеются системы *типов данных.* Бестиповые языки, такие как языки ассемблера или простейшее λ-исчисление, встречаются редко.

Типы в языках программирования

- В большинстве языков программирования имеются системы *типов данных*. Бестиповые языки, такие как языки ассемблера или простейшее λ-исчисление, встречаются редко.
- В бестиповом языке любую операцию можно совершить над любыми данными. Дисциплина типов данных налагает определённые *ограничения* на применение операций (функций), чтобы отсечь *бессмысленные* ошибочные применения.

Типы в языках программирования

- В большинстве языков программирования имеются системы *типов данных.* Бестиповые языки, такие как языки ассемблера или простейшее λ-исчисление, встречаются редко.
- В бестиповом языке любую операцию можно совершить над любыми данными. Дисциплина типов данных налагает определённые *ограничения* на применение операций (функций), чтобы отсечь *бессмысленные* ошибочные применения.
    - Например, выражение `2+2` осмысленно (хотя, может быть, вычисляет не то, что нам на самом деле нужно), а выражение `2+"two"` скорее всего бессмысленно.

Типы в языках программирования

- Таким образом, система типов выполняет охранительную функцию: проверки корректности типов запрещают некоторые конструкции («мешают программировать»).

- Таким образом, система типов выполняет охранительную функцию: проверки корректности типов запрещают некоторые конструкции («мешают программировать»).
- При этом эти конструкции не всегда совершенно бессмысленные. Например, комбинатор неподвижной точки $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ скорее всего будет некорректен с точки зрения системы типов (аргумент функции не может иметь тот же тип, что и сама функция), однако разумно используется для реализации рекурсии.

Типы в языках программирования

- Таким образом, система типов выполняет охранительную функцию: проверки корректности типов запрещают некоторые конструкции («мешают программировать»).
- При этом эти конструкции не всегда совершенно бессмысленные. Например, комбинатор неподвижной точки $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ скорее всего будет некорректен с точки зрения системы типов (аргумент функции не может иметь тот же тип, что и сама функция), однако разумно используется для реализации рекурсии.
- Для собственно исполнения программы (вычисления) типы обыкновенно не нужны.

Типы в языках программирования

- С другой стороны, контроль типов помогает избежать многих ошибок при программировании.

Типы в языках программирования

- С другой стороны, контроль типов помогает избежать многих ошибок при программировании.
    - Фактически, контроль типов — это начальный элемент *верификации* (формального доказательства) корректности работы программы.

Типы в языках программирования

- С другой стороны, контроль типов помогает избежать многих ошибок при программировании.
    - Фактически, контроль типов — это начальный элемент *верификации* (формального доказательства) корректности работы программы.
    - Используя развитую систему типов *(зависимые типы)*, можно свести задачу верификации к проверке типов. Например, вместо $\mathrm{mod} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ можно потребовать более точный тип

$$\mathrm{mod}' : ((x, y) : \mathbb{N} \times \mathbb{N}) \mapsto$$
$$\mapsto r : \{r : \mathbb{N} \mid y = 0 \vee \exists q : \mathbb{N}(x = y \cdot q + r \wedge r < y)\},$$

Типы в языках программирования

- С другой стороны, контроль типов помогает избежать многих ошибок при программировании.
    - Фактически, контроль типов — это начальный элемент *верификации* (формального доказательства) корректности работы программы.
    - Используя развитую систему типов *(зависимые типы)*, можно свести задачу верификации к проверке типов. Например, вместо $\mathrm{mod} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ можно потребовать более точный тип

$$\begin{aligned}\mathrm{mod}' : ((x, y) : \mathbb{N} \times \mathbb{N}) \mapsto \\ \mapsto r : \{r : \mathbb{N} \mid y = 0 \vee \exists q : \mathbb{N}(x = y \cdot q + r \wedge r < y)\},\end{aligned}$$

    - Такие возможности есть в Coq, Agda и проч.

Типы в языках программирования

- Типы также используются как косвенный способ документирования программного кода: по типу функции зачастую можно понять, что она делает.

Типы в языках программирования

- Типы также используются как косвенный способ документирования программного кода: по типу функции зачастую можно понять, что она делает.
  - Например, из типа $\mathbf{B} : (B \to C) \to ((A \to B) \to (A \to C))$ даже без реализации ($\mathbf{B} = \lambda fgx.f(gx)$) понятно, что $\mathbf{B}$ реализует композицию функций.

Типы в языках программирования

- Типы также используются как косвенный способ документирования программного кода: по типу функции зачастую можно понять, что она делает.
    - Например, из типа $\mathbf{B} : (B \to C) \to ((A \to B) \to (A \to C))$ даже без реализации ($\mathbf{B} = \lambda fgx.f(gx)$) понятно, что $\mathbf{B}$ реализует композицию функций.
    - Более того, если это полиморфный тип, где A, B, C — абстрактные переменные, то можно *доказать*, что $\mathbf{B}$ — это оператор композиции. Это одна из так называемых *free theorems*.

Типы в языках программирования

- Типы также используются как косвенный способ документирования программного кода: по типу функции зачастую можно понять, что она делает.
  - Например, из типа $\mathbf{B} : (B \to C) \to ((A \to B) \to (A \to C))$ даже без реализации ($\mathbf{B} = \lambda f g x. f(gx)$) понятно, что $\mathbf{B}$ реализует композицию функций.
  - Более того, если это полиморфный тип, где A, B, C — абстрактные переменные, то можно *доказать*, что $\mathbf{B}$ — это оператор композиции. Это одна из так называемых *free theorems.*
- Наконец, типы влияют на исполнение кода при так называемом *ad hoc полиморфизме*, или *перегрузке* функции. Пример (работает в C++, но не в C):

```
void f(int x) { printf("integer\n"); }
void f(char x) { printf("character\n"); }
```

Птицы и комбинаторы

- $\mathbf{B} = \lambda fgx.f(gx)$ — это один из *комбинаторов.*

Птицы и комбинаторы

- $\mathbf{B} = \lambda fgx.f(gx)$ — это один из *комбинаторов.*
- Комбинаторами называется замкнутые (без свободных переменных) термы чистого λ-исчисления. Они выражают абстрактные алгоритмы работы с функциями и имеют, как мы увидим, параметрически полиморфные типы.

Птицы и комбинаторы

- $\mathbf{B} = \lambda fgx.f(gx)$ — это один из *комбинаторов.*
- Комбинаторами называется замкнутые (без свободных переменных) термы чистого λ-исчисления. Они выражают абстрактные алгоритмы работы с функциями и имеют, как мы увидим, параметрически полиморфные типы.
- Комбинаторы, следуя Смаллиану (“To mock a mockingbird”), называются по первым буквам названий видов птиц.

Птицы и комбинаторы

- $\mathbf{B} = \lambda fgx.f(gx)$ — это один из *комбинаторов.*
- Комбинаторами называется замкнутые (без свободных переменных) термы чистого λ-исчисления. Они выражают абстрактные алгоритмы работы с функциями и имеют, как мы увидим, параметрически полиморфные типы.
- Комбинаторы, следуя Смаллиану (“To mock a mockingbird”), называются по первым буквам названий видов птиц.
- **B** — сиалия (bluebird), семейства дроздовых.

Sialia currucoides

Особенности типизации в Haskell'e

- Типизация — это процесс присвоения типов объектам (переменным и составным выражениям) для дальнейшего контроля типов.

Особенности типизации в Haskell'e

- Типизация — это процесс присвоения типов объектам (переменным и составным выражениям) для дальнейшего контроля типов.
- Типизация в разных языках программирования устроена по-разному.

Особенности типизации в Haskell'е

- Типизация — это процесс присвоения типов объектам (переменным и составным выражениям) для дальнейшего контроля типов.
- Типизация в разных языках программирования устроена по-разному.
- Перечислим основные свойства типизации в Haskell'е.

Особенности типизации в Haskell’е

1. Система типов достаточно богатая, в частности, присутствуют функциональные *(«стрельчатые»)* типы вида $A \to B$.

Особенности типизации в Haskell’е

1. Система типов достаточно богатая, в частности, присутствуют функциональные *(«стрельчатые»)* типы вида $A \to B$.
2. Типизация *сильная* (или строгая, strong): корректность типов контролируется последовательно, её нельзя «обойти» — как, например, приведением к типу `void*` в C.

Особенности типизации в Haskell'e

1. Система типов достаточно богатая, в частности, присутствуют функциональные *(«стрельчатые»)* типы вида $A \to B$.
2. Типизация *сильная* (или строгая, strong): корректность типов контролируется последовательно, её нельзя «обойти» — как, например, приведением к типу `void*` в C.
3. При этом имеется развитый *полиморфизм* (о нём мы поговорим позже).

Особенности типизации в Haskell'e

1. Система типов достаточно богатая, в частности, присутствуют функциональные *(«стрельчатые»)* типы вида $A \to B$.
2. Типизация *сильная* (или строгая, strong): корректность типов контролируется последовательно, её нельзя «обойти» — как, например, приведением к типу `void*` в C.
3. При этом имеется развитый *полиморфизм* (о нём мы поговорим позже).
4. *Статическая* типизация: проверки типов выполняются на этапе компиляции, при выполнении типы не имеют значения. (Противоположность: *динамическая* типизация, когда типы вычисляются и проверяются при исполнении.)

Особенности типизации в Haskell'е

1. Система типов достаточно богатая, в частности, присутствуют функциональные *(«стрельчатые»)* типы вида $A \to B$.
2. Типизация *сильная* (или строгая, strong): корректность типов контролируется последовательно, её нельзя «обойти» — как, например, приведением к типу `void*` в C.
3. При этом имеется развитый *полиморфизм* (о нём мы поговорим позже).
4. *Статическая* типизация: проверки типов выполняются на этапе компиляции, при выполнении типы не имеют значения. (Противоположность: *динамическая* типизация, когда типы вычисляются и проверяются при исполнении.)
    - Динамические типы: Data.Dynamic.

Особенности типизации в Haskell’е

5. Типизация может быть *неявной:* программист может не указывать типы, и тогда они будут автоматически вычислены (выведены) с помощью *алгоритма выведения типов* (type inference).

5. Типизация может быть *неявной:* программист может не указывать типы, и тогда они будут автоматически вычислены (выведены) с помощью *алгоритма выведения типов* (type inference). Явное указание типов также допускается.

Особенности типизации в Haskell'е

5. Типизация может быть *неявной:* программист может не указывать типы, и тогда они будут автоматически вычислены (выведены) с помощью *алгоритма выведения типов* (type inference). Явное указание типов также допускается.

- Выведение типов неразрывно связано с полиморфизмом. Если одно и то же выражение можно типизовать по-разному (т.е. оно является полиморфным), то алгоритм выведения типов должен выбрать в некотором смысле *наиболее общий* (наиболее абстрактный) тип.

Особенности типизации в Haskell'e

5. Типизация может быть *неявной:* программист может не указывать типы, и тогда они будут автоматически вычислены (выведены) с помощью *алгоритма выведения типов* (type inference). Явное указание типов также допускается.

- Выведение типов неразрывно связано с полиморфизмом. Если одно и то же выражение можно типизовать по-разному (т.е. оно является полиморфным), то алгоритм выведения типов должен выбрать в некотором смысле *наиболее общий* (наиболее абстрактный) тип.
- Система типов в Haskell'e и алгоритм выведения типов основаны на *системе Хиндли – Милнера,* о которой мы поговорим позже.

Параметрический полиморфизм

- Haskell поддерживает два вида полиморфизма: *ad hoc* (аналог перегрузки функций в C++, реализуется через *классы типов*) и *параметрический* (в состав типа могут входить *переменные*, вместо которых можно подставить произвольный тип или тип из какого-то класса).

Параметрический полиморфизм

- Haskell поддерживает два вида полиморфизма: *ad hoc* (аналог перегрузки функций в C++, реализуется через *классы типов*) и *параметрический* (в состав типа могут входить *переменные*, вместо которых можно подставить произвольный тип или тип из какого-то класса).
- В C++ параметрический полиморфизм реализуется с помощью механизма *шаблонов* (templates).

Параметрический полиморфизм

- Haskell поддерживает два вида полиморфизма: *ad hoc* (аналог перегрузки функций в C++, реализуется через *классы типов*) и *параметрический* (в состав типа могут входить *переменные*, вместо которых можно подставить произвольный тип или тип из какого-то класса).
- В C++ параметрический полиморфизм реализуется с помощью механизма *шаблонов* (templates).
- Мы будем обсуждать параметрический полиморфизм.

Параметрический полиморфизм

- Параметрический полиморфизм проще всего проиллюстрировать на комбинаторах.

Параметрический полиморфизм

- Параметрический полиморфизм проще всего проиллюстрировать на комбинаторах.
- Рассмотрим комбинатор $\mathbf{K} = \lambda x.\lambda y.x$.

Параметрический полиморфизм

- Параметрический полиморфизм проще всего проиллюстрировать на комбинаторах.
- Рассмотрим комбинатор $\mathbf{K} = \lambda x.\lambda y.x$.
- Птица — пустельга (kestrel).

Falco tinnunculus

Параметрический полиморфизм

- Комбинатор $\mathbf{K} = \lambda x.\lambda y.x$ (Haskell: `\x y -> x`) берёт два аргумента и возвращает первый.

Параметрический полиморфизм

- Комбинатор $\mathbf{K} = \lambda x.\lambda y.x$ (Haskell: `\x y -> x`) берёт два аргумента и возвращает первый.
- При этом типы аргументов могут быть разными, а сам $\mathbf{K}$ может быть типизован и как $\mathtt{Int} \to (\mathtt{Int} \to \mathtt{Int})$, и как $\mathtt{Char} \to (\mathtt{Bool} \to \mathtt{Char})$, и даже как $(\mathtt{Int} \to \mathtt{Bool}) \to (\mathtt{Char} \to (\mathtt{Int} \to \mathtt{Bool}))$.

Параметрический полиморфизм

- Комбинатор **K** $= \lambda x.\lambda y.x$ (Haskell: `\x y -> x`) берёт два аргумента и возвращает первый.
- При этом типы аргументов могут быть разными, а сам **K** может быть типизован и как $\texttt{Int} \to (\texttt{Int} \to \texttt{Int})$, и как $\texttt{Char} \to (\texttt{Bool} \to \texttt{Char})$, и даже как $(\texttt{Int} \to \texttt{Bool}) \to (\texttt{Char} \to (\texttt{Int} \to \texttt{Bool}))$.
- В языке без полиморфизма пришлось бы программировать каждую версию **K** отдельно:

```
int K_int(int x, int y) { return x; }
int K_charint(char x, int y) { return x; }
```

Параметрический полиморфизм

- Комбинатор $\mathbf{K} = \lambda x.\lambda y.x$ (Haskell: `\x y -> x`) берёт два аргумента и возвращает первый.
- При этом типы аргументов могут быть разными, а сам **K** может быть типизован и как $\mathtt{Int} \to (\mathtt{Int} \to \mathtt{Int})$, и как $\mathtt{Char} \to (\mathtt{Bool} \to \mathtt{Char})$, и даже как $(\mathtt{Int} \to \mathtt{Bool}) \to (\mathtt{Char} \to (\mathtt{Int} \to \mathtt{Bool}))$.
- В языке без полиморфизма пришлось бы программировать каждую версию **K** отдельно:

```
int K_int(int x, int y) { return x; }
int K_charint(char x, int y) { return x; }
```

  - С перегрузкой (ad hoc полиморфизм) эти функции можно было бы назвать одним словом, но дублирования кода не избежать.

Параметрический полиморфизм

- В Haskell'е комбинатор **K** получает (автоматически, с помощью выведения типов) абстрактный тип

$$p_1 \to (p_2 \to p_1),$$

где p_1 и p_2 — *переменные по типам.*

Параметрический полиморфизм

- В Haskell'е комбинатор **K** получает (автоматически, с помощью выведения типов) абстрактный тип

$$p_1 \to (p_2 \to p_1),$$

  где p_1 и p_2 — *переменные по типам.*
- Этот параметрический тип является *наиболее общим* в том смысле, что любой другой корректный тип для **K** получается из $p_1 \to (p_2 \to p_1)$, подстановкой конкретных типов вместо p_1 и p_2.

Параметрический полиморфизм

- В Haskell'е комбинатор **K** получает (автоматически, с помощью выведения типов) абстрактный тип

$$p_1 \to (p_2 \to p_1),$$

  где p_1 и p_2 — *переменные по типам.*
- Этот параметрический тип является *наиболее общим* в том смысле, что любой другой корректный тип для **K** получается из $p_1 \to (p_2 \to p_1)$, подстановкой конкретных типов вместо p_1 и p_2.
- Реализация в C++ с помощью шаблонов:

```
template<typename P1, typename P2>
  P1 kestrel(P1 x, P2 y) { return x; }
```

Параметрический полиморфизм

- В Haskell'е значения параметров (переменных по типам) могут ограничиваться классами типов, например:

  ```
  (\x y z -> x (y+z)) :: Num t1 => (t1 -> t2) -> t1
    -> t1 -> t2
  ```

Параметрический полиморфизм

- В Haskell'е значения параметров (переменных по типам) могут ограничиваться классами типов, например:
  `(\x y z -> x (y+z)) :: Num t1 => (t1 -> t2) -> t1 -> t1 -> t2`
- Таким образом, параметрический полиморфизм может сочетаться с ad hoc полиморфизмом: реализация операции `(+) :: Num t1 => t1 -> t1 -> t1` зависит от типа `t1`.

Параметрический полиморфизм

- В Haskell'е значения параметров (переменных по типам) могут ограничиваться классами типов, например:
  ```
  (\x y z -> x (y+z)) :: Num t1 => (t1 -> t2) -> t1
    -> t1 -> t2
  ```
- Таким образом, параметрический полиморфизм может сочетаться с ad hoc полиморфизмом: реализация операции `(+) :: Num t1 => t1 -> t1 -> t1` зависит от типа `t1`.
- Мы начнём с простой типизации «чистого» λ-исчисления, где переменные по типам всегда могут принимать произвольные значения.

$\lambda_{\to}$: простое типизованное λ-исчисление

- Множество типов строится из переменных по типам $p_1, p_2, p_3, \ldots$ (не путать с переменными по объектам $x, y, z, \ldots$) с помощью единственной операции $\to$. Если A и B — типы, то $(A \to B)$ — тоже тип.

$\lambda_{\to}$: простое типизованное λ-исчисление

- Множество типов строится из переменных по типам $p_1, p_2, p_3, \ldots$ (не путать с переменными по объектам $x, y, z, \ldots$) с помощью единственной операции $\to$. Если A и B — типы, то $(A \to B)$ — тоже тип.
    - Константных типов (вроде `Bool`, `Int`) нет, как нет и констант-термов.

$\lambda_\to$: простое типизованное λ-исчисление

- Множество типов строится из переменных по типам $p_1, p_2, p_3, \ldots$ (не путать с переменными по объектам $x, y, z, \ldots$) с помощью единственной операции $\to$. Если A и B — типы, то $(A \to B)$ — тоже тип.
  - Константных типов (вроде `Bool`, `Int`) нет, как нет и констант-термов.
- Единственное *ограничение типизации:* применение (uv) корректно только тогда, когда v имеет тип A, а u имеет тип $(A \to B)$ для некоторых типов A и B. В этом случае (uv) имеет тип B.

$\lambda_\to$: простое типизованное λ-исчисление

- Множество типов строится из переменных по типам $p_1, p_2, p_3, \dots$ (не путать с переменными по объектам $x, y, z, \dots$) с помощью единственной операции $\to$. Если A и B — типы, то $(A \to B)$ — тоже тип.
  - Константных типов (вроде `Bool`, `Int`) нет, как нет и констант-термов.
- Единственное *ограничение типизации:* применение (uv) корректно только тогда, когда v имеет тип A, а u имеет тип $(A \to B)$ для некоторых типов A и B. В этом случае (uv) имеет тип B.
- λ-абстракция может применяться всегда. При этом если переменная x имеет тип A, а терм u — тип B, то $\lambda x.u$ имеет тип $(A \to B)$.

Типизация переменных

- Осталось разобраться, как присваивать типы переменным.

Типизация переменных

- Осталось разобраться, как присваивать типы переменным.
  - **Внимание:** тип переменной-объекта не обязательно является переменной-типом. Например, в типизации комбинатора $\mathbf{B} = \lambda fgx.f(gx)$ переменные f и g имеют сложные типы $(B \to C)$ и $(A \to B)$.

Типизация переменных

- Осталось разобраться, как присваивать типы переменным.
  - **Внимание:** тип переменной-объекта не обязательно является переменной-типом. Например, в типизации комбинатора $\mathbf{B} = \lambda fgx.f(gx)$ переменные f и g имеют сложные типы $(B \to C)$ и $(A \to B)$.
- Переменные бывают свободные и связанные (находящиеся под λ'ми).

Типизация переменных

- Осталось разобраться, как присваивать типы переменным.
  - **Внимание:** тип переменной-объекта не обязательно является переменной-типом. Например, в типизации комбинатора $\mathbf{B} = \lambda fgx.f(gx)$ переменные f и g имеют сложные типы $(B \to C)$ и $(A \to B)$.
- Переменные бывают свободные и связанные (находящиеся под λ'ми).
- Для простоты будем считать, что множества свободных и связанных переменных не пересекаются (иначе применим α-преобразования).

Типизация переменных

- Осталось разобраться, как присваивать типы переменным.
  - **Внимание:** тип переменной-объекта не обязательно является переменной-типом. Например, в типизации комбинатора $\mathbf{B} = \lambda fgx.f(gx)$ переменные f и g имеют сложные типы $(B \to C)$ и $(A \to B)$.
- Переменные бывают свободные и связанные (находящиеся под λ'ми).
- Для простоты будем считать, что множества свободных и связанных переменных не пересекаются (иначе применим α-преобразования).
- Типы свободных переменных декларируются явно в *контексте* $\Gamma = x_1 : A_1, \ldots, x_n : A_n$.

Типизация по Чёрчу

- Для типизации связанных переменных есть два подхода.

Типизация по Чёрчу

- Для типизации связанных переменных есть два подхода.
- При типизации *по Чёрчу* («жёсткой»), при каждой λ'е явно указывается тип соответствующей переменной.

Типизация по Чёрчу

- Для типизации связанных переменных есть два подхода.
- При типизации *по Чёрчу* («жёсткой»), при каждой λ'е явно указывается тип соответствующей переменной.
- Далее тип каждого терма вычисляется однозначно.

Типизация по Чёрчу

- Для типизации связанных переменных есть два подхода.
- При типизации *по Чёрчу* («жёсткой»), при каждой λ'е явно указывается тип соответствующей переменной.
- Далее тип каждого терма вычисляется однозначно.
- Типизация по Чёрчу приводит к необходимости изменения языка термов (добавить указания типов), а также фактически к отказу от полиморфизма. Так, вместо одного комбинатора $\mathbf{B} = \lambda fgx.f(gx)$ нужно ввести отдельный комбинатор

$$\mathbf{B}_{A,B,C} = \lambda f^{B\to C}.\lambda g^{A\to B}.\lambda x^{A}.f(gx)$$

для каждого набора типов A, B, C.

Типизация по Карри

- Более гибкой является *типизация по Карри.*

Типизация по Карри

- Более гибкой является *типизация по Карри.*
- При типизации по Карри данный терм в данном контексте может иметь много различных *возможных* типов.

Типизация по Карри

- Более гибкой является *типизация по Карри.*
- При типизации по Карри данный терм в данном контексте может иметь много различных *возможных* типов.
- Запись $\Gamma \vdash u : B$ означает что B — *один* из допустимых типов для u в контексте Γ (т.е. что связанным переменным в u *можно* приписать такие типы, что полученный терм будет корректно типизован по Чёрчу, и его типом будет B).

Типизация по Карри

- Более гибкой является *типизация по Карри.*
- При типизации по Карри данный терм в данном контексте может иметь много различных *возможных* типов.
- Запись $\Gamma \vdash u : B$ означает что B — *один* из допустимых типов для u в контексте Γ (т.е. что связанным переменным в u *можно* приписать такие типы, что полученный терм будет корректно типизован по Чёрчу, и его типом будет B).
- Запись $\Gamma \vdash u : B$ называется *утверждением о типизуемости.* Такие утверждения будут *доказываться как теоремы* в специально построенном логическом исчислении.

Типизация по Карри

- Более гибкой является *типизация по Карри.*
- При типизации по Карри данный терм в данном контексте может иметь много различных *возможных* типов.
- Запись $\Gamma \vdash u : B$ означает что B — *один* из допустимых типов для u в контексте Γ (т.е. что связанным переменным в u *можно* приписать такие типы, что полученный терм будет корректно типизован по Чёрчу, и его типом будет B).
- Запись $\Gamma \vdash u : B$ называется *утверждением о типизуемости.* Такие утверждения будут *доказываться как теоремы* в специально построенном логическом исчислении.
- Терм u не типизуем в контексте Γ, если $\Gamma \vdash u : B$ не доказуемо (не имеет места) ни для какого B.

Типизация по Карри

- Правила исчисления для типизации по Карри соответствуют правилам построения термов:

$$\frac{}{\Gamma, x : A \vdash x : A}\ \text{Ax} \qquad \frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash (\lambda x.u) : (A \to B)}\ \text{Abs}$$

$$\frac{\Gamma \vdash u : (A \to B) \quad \Gamma \vdash v : A}{\Gamma \vdash (uv) : B}\ \text{App}$$

Типизация по Карри

- Правила исчисления для типизации по Карри соответствуют правилам построения термов:

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathrm{Ax} \qquad \frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash (\lambda x.u) : (A \to B)}\ \mathrm{Abs}$$

$$\frac{\Gamma \vdash u : (A \to B) \quad \Gamma \vdash v : A}{\Gamma \vdash (uv) : B}\ \mathrm{App}$$

- Доказательство (вывод) удобно представлять в виде дерева, где в корне стоит целевое утверждение $\Gamma \vdash u : B$, в листьях — аксиомы (Ax), а внутренние вершины соответствуют правилам App и Abs.

Типизация по Карри

- Введённая нами система типов обладает свойством *безопасности типов* относительно β-редукции: если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.

Типизация по Карри

- Введённая нами система типов обладает свойством *безопасности типов* относительно β-редукции: если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.
- Это означает, что в процессе вычислений можно не контролировать типы, достаточно (статической) проверки в начале.

Типизация по Карри

- Введённая нами система типов обладает свойством *безопасности типов* относительно β-редукции: если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.
- Это означает, что в процессе вычислений можно не контролировать типы, достаточно (статической) проверки в начале.
- В обратную сторону, однако, это не работает: если $u \to_\beta u'$ и $\Gamma \vdash u' : B$, то не обязательно $\Gamma \vdash u : B$. Может оказаться, что u вообще не типизируем, либо у него меньше корректных типов, чем у u'.

Типизация по Карри

- Введённая нами система типов обладает свойством *безопасности типов* относительно β-редукции: если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.
- Это означает, что в процессе вычислений можно не контролировать типы, достаточно (статической) проверки в начале.
- В обратную сторону, однако, это не работает: если $u \to_\beta u'$ и $\Gamma \vdash u' : B$, то не обязательно $\Gamma \vdash u : B$. Может оказаться, что u вообще не типизируем, либо у него меньше корректных типов, чем у u'.
    - **Задача.** Придумать конкретные примеры.

Типизация по Карри

- Пример типизации комбинатора **B**:

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        f : (B \to C), \ldots \vdash f : (B \to C)
        \qquad
        \cfrac{
          \ldots, g : (A \to B), \ldots \vdash g : (A \to B)
          \qquad
          \ldots, x : A \vdash x : A
        }{
          f : (B \to C), g : (A \to B), x : A \vdash gx : B
        }\ \text{App}
      }{
        f : (B \to C), g : (A \to B), x : A \vdash f(gx) : C
      }\ \text{App}
    }{
      f : (B \to C), g : (A \to B) \vdash \lambda x.f(gx) : A \to C
    }\ \text{Abs}
  }{
    f : (B \to C) \vdash \lambda g.\lambda x.f(gx) : (A \to B) \to (A \to C)
  }\ \text{Abs}
}{
  \vdash \lambda f.\lambda g.\lambda x.f(gx) : (B \to C) \to ((A \to B) \to (A \to C))
}\ \text{Abs}
$$

- Пример типизации комбинатора **B**:

$$
\dfrac{
  \dfrac{
    f : (B \to C), \ldots \vdash f : (B \to C)
    \qquad
    \dfrac{
      \ldots, g : (A \to B), \ldots \vdash g : (A \to B)
      \qquad
      \ldots, x : A \vdash x : A
    }{
      f : (B \to C), g : (A \to B), x : A \vdash gx : B
    }\ \text{App}
  }{
    \dfrac{
      \dfrac{
        f : (B \to C), g : (A \to B), x : A \vdash f(gx) : C
      }{
        f : (B \to C), g : (A \to B) \vdash \lambda x.f(gx) : A \to C
      }\ \text{Abs}
    }{
      f : (B \to C) \vdash \lambda g.\lambda x.f(gx) : (A \to B) \to (A \to C)
    }\ \text{Abs}
  }\ \text{App}
}{
  \vdash \lambda f.\lambda g.\lambda x.f(gx) : (B \to C) \to ((A \to B) \to (A \to C))
}\ \text{Abs}
$$

- Этот вывод показывает, что терм $\lambda f.\lambda g.\lambda x.f(gx)$ типизуем типом $(B \to C) \to ((A \to B) \to (A \to C))$ для *произвольных* типов A, B, C.

Типизация по Карри

- Пример типизации комбинатора **B**:

$$
\dfrac{\dfrac{\dfrac{\dfrac{f : (B \to C), \ldots \vdash f : (B \to C) \qquad \dfrac{\ldots, g : (A \to B), \ldots \vdash g : (A \to B) \qquad \ldots, x : A \vdash x : A}{f : (B \to C), g : (A \to B), x : A \vdash gx : B}\ \text{App}}{f : (B \to C), g : (A \to B), x : A \vdash f(gx) : C}\ \text{App}}{f : (B \to C), g : (A \to B) \vdash \lambda x.f(gx) : A \to C}\ \text{Abs}}{f : (B \to C) \vdash \lambda g.\lambda x.f(gx) : (A \to B) \to (A \to C)}\ \text{Abs}}{\vdash \lambda f.\lambda g.\lambda x.f(gx) : (B \to C) \to ((A \to B) \to (A \to C))}\ \text{Abs}
$$

- Этот вывод показывает, что терм $\lambda f.\lambda g.\lambda x.f(gx)$ типизуем типом $(B \to C) \to ((A \to B) \to (A \to C))$ для *произвольных* типов A, B, C.
- Иначе говоря, корректным типом для **B** (при пустом контексте) будет $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$ с любой подстановкой типов вместо переменных r_1, r_2, r_3.

Наиболее общий тип

- Оказывается (мы это докажем), что это **полное** описание всех типов для **B**.

Наиболее общий тип

- Оказывается (мы это докажем), что это **полное** описание всех типов для $\mathbf{B}$.
- А именно, если $\vdash \mathbf{B} : T$, то T получается подстановкой из $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$.

Наиболее общий тип

- Оказывается (мы это докажем), что это **полное** описание всех типов для $\mathbf{B}$.
- А именно, если $\vdash \mathbf{B} : T$, то T получается подстановкой из $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$.
- Сам $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$ называется *наиболее общим типом* для $\mathbf{B}$.

Наиболее общий тип

- Оказывается (мы это докажем), что это **полное** описание всех типов для $\mathbf{B}$.
- А именно, если $\vdash \mathbf{B} : T$, то T получается подстановкой из $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$.
- Сам $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$ называется *наиболее общим типом* для $\mathbf{B}$.
- Более того, оказывается, что так будет всегда!

Наиболее общий тип

- Оказывается (мы это докажем), что это **полное** описание всех типов для **B**.
- А именно, если $\vdash \mathbf{B} : T$, то T получается подстановкой из $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$.
- Сам $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$ называется *наиболее общим типом* для **B**.
- Более того, оказывается, что так будет всегда!
- Любой терм u либо вообще не типизуем в контексте Γ, либо имеет наиболее общий тип, из которого все остальные получаются подстановкой типов вместо переменных, не встречающихся в Γ.

Наиболее общий тип

- Оказывается (мы это докажем), что это **полное** описание всех типов для $\mathbf{B}$.
- А именно, если $\vdash \mathbf{B} : T$, то T получается подстановкой из $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$.
- Сам $(r_2 \to r_3) \to ((r_1 \to r_2) \to (r_1 \to r_3))$ называется *наиболее общим типом* для $\mathbf{B}$.
- Более того, оказывается, что так будет всегда!
- Любой терм u либо вообще не типизуем в контексте Γ, либо имеет наиболее общий тип, из которого все остальные получаются подстановкой типов вместо переменных, не встречающихся в Γ.
- «Неизменяемые» переменные по типам, используемые в контексте Γ, обозначим через $p_1, p_2, \ldots$. Остальные — $r_1, r_2, \ldots$

Наиболее общий тип

- В Haskell'е используется более мощная система типов, основанная на типизации Хиндли – Милнера (об этом мы поговорим на следующих лекциях).

Наиболее общий тип

- В Haskell'е используется более мощная система типов, основанная на типизации Хиндли – Милнера (об этом мы поговорим на следующих лекциях).
- В GHCi вычислить наиболее общий тип λ-терма можно командой `:t`

Наиболее общий тип

- В Haskell'е используется более мощная система типов, основанная на типизации Хиндли – Милнера (об этом мы поговорим на следующих лекциях).
- В GHCi вычислить наиболее общий тип λ-терма можно командой `:t`
- Например, для нумералов Чёрча `:t (\s o -> s (s (s o)))` даёт

```
(\s o -> s (s (s o))) :: (t -> t) -> t -> t
```

Наиболее общий тип

- В Haskell'е используется более мощная система типов, основанная на типизации Хиндли – Милнера (об этом мы поговорим на следующих лекциях).
- В GHCi вычислить наиболее общий тип λ-терма можно командой `:t`
- Например, для нумералов Чёрча `:t (\s o -> s (s (s o)))` даёт

  `(\s o -> s (s (s o))) :: (t -> t) -> t -> t`
- При этом для, например, операции сложения `churchPlus = \x y s o -> x s (y s o)` тип оказывается более общим, чем ожидалось: `(t1 -> t2 -> t3) -> (t1 -> t4 -> t2) -> t1 -> t4 -> t3`, а не `((t -> t) -> t -> t) -> ((t -> t) -> t -> t) -> (t -> t) -> t -> t`.

Типы и рекурсия

- Как мы видим, нумералы Чёрча, а также реализованные в чистом λ-исчислении булевы операции, типизируемы по Карри.

Типы и рекурсия

- Как мы видим, нумералы Чёрча, а также реализованные в чистом λ-исчислении булевы операции, типизируемы по Карри.
- Однако это не так для комбинатора неподвижной точки $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$. Действительно, уже подтерм xx не пройдёт контроль типов, т.к. $(A \to B) \neq A$.

- Как мы видим, нумералы Чёрча, а также реализованные в чистом λ-исчислении булевы операции, типизируемы по Карри.
- Однако это не так для комбинатора неподвижной точки $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$. Действительно, уже подтерм xx не пройдёт контроль типов, т.к. $(A \to B) \neq A$.
    - На самом деле, никакой другой комбинатор неподвижной точки типизовать тоже нельзя, поскольку в чистом λ-исчислении все типизуемые термы сильно нормализуемы.

Типы и рекурсия

- Как мы видим, нумералы Чёрча, а также реализованные в чистом λ-исчислении булевы операции, типизируемы по Карри.
- Однако это не так для комбинатора неподвижной точки $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$. Действительно, уже подтерм xx не пройдёт контроль типов, т.к. $(A \to B) \neq A$.
  - На самом деле, никакой другой комбинатор неподвижной точки типизовать тоже нельзя, поскольку в чистом λ-исчислении все типизуемые термы сильно нормализуемы.
- Однако $\mathbf{Y}$ как «чёрный ящик» типизуем! Можно ввести константу $\mathbb{Y}$ полиморфного типа $(r \to r) \to r$ и редукцию $\mathbb{Y}u \to_\delta u(\mathbb{Y}u)$.

$\lambda_{\rightarrow}$Y-исчисление

- В λ-исчислении с простой системой типов, расширенном константой $\mathbb{Y}$ и δ-редукцией, можно реализовать рекурсию, используя только типизуемые термы. Значит, это опять полный по Тьюрингу язык.

$\lambda_{\to}$Y-исчисление

- В λ-исчислении с простой системой типов, расширенном константой $\mathbb{Y}$ и δ-редукцией, можно реализовать рекурсию, используя только типизуемые термы. Значит, это опять полный по Тьюрингу язык.
- Вместо константы $\mathbb{Y}$ можно ввести оператор Y, связывающий переменную: $\mathrm{Y}x.u$, и получить исчисление $\lambda_{\to}\mathrm{Y}$.

$\lambda_{\to}$Y-исчисление

- В λ-исчислении с простой системой типов, расширенном константой $\mathbb{Y}$ и δ-редукцией, можно реализовать рекурсию, используя только типизуемые термы. Значит, это опять полный по Тьюрингу язык.
- Вместо константы $\mathbb{Y}$ можно ввести оператор Y, связывающий переменную: $\mathrm{Y}x.u$, и получить исчисление $\lambda_{\to}$Y.
    - Типизация:

$$\frac{\Gamma, x : B \vdash u : B}{\Gamma \vdash (\mathrm{Y}x.u) : B}\ \mathrm{Fix}$$

- В λ-исчислении с простой системой типов, расширенном константой $\mathbb{Y}$ и δ-редукцией, можно реализовать рекурсию, используя только типизуемые термы. Значит, это опять полный по Тьюрингу язык.
- Вместо константы $\mathbb{Y}$ можно ввести оператор Y, связывающий переменную: $\mathrm{Y}x.u$, и получить исчисление $\lambda_{\to}\mathrm{Y}$.
    - Типизация:

$$\frac{\Gamma, x : B \vdash u : B}{\Gamma \vdash (\mathrm{Y}x.u) : B}\ \mathrm{Fix}$$

    - δ-редукция: $\mathrm{Y}x.u \to_{\delta} u[x := \mathrm{Y}x.u]$

- В λ-исчислении с простой системой типов, расширенном константой $\mathbb{Y}$ и δ-редукцией, можно реализовать рекурсию, используя только типизуемые термы. Значит, это опять полный по Тьюрингу язык.
- Вместо константы $\mathbb{Y}$ можно ввести оператор Y, связывающий переменную: $\mathrm{Y}x.u$, и получить исчисление $\lambda_{\to}$Y.
  - Типизация:
  $$\frac{\Gamma, x : B \vdash u : B}{\Gamma \vdash (\mathrm{Y}x.u) : B}\ \mathrm{Fix}$$
  - δ-редукция: $\mathrm{Y}x.u \to_\delta u[x := \mathrm{Y}x.u]$
  - С помощью $\mathbb{Y}$ выражается так: $\mathrm{Y}x.u = \mathbb{Y}(\lambda x.u)$; тогда $\mathbb{Y}(\lambda x.u) \to_\delta (\lambda x.u)(\mathbb{Y}(\lambda x.u)) \to_\beta u[x := \mathbb{Y}(\lambda x.u)]$.

Выведение типов

- Задача поиска наиболее общего типа (или выяснения, что терм нетипизуем) называется задачей *выведения типа* (type inference).

Выведение типов

- Задача поиска наиболее общего типа (или выяснения, что терм нетипизуем) называется задачей *выведения типа* (type inference).
- Эта задача алгоритмически разрешима, и соответствующий алгоритм реализован в GHC.

Выведение типов

- Задача поиска наиболее общего типа (или выяснения, что терм нетипизуем) называется задачей *выведения типа* (type inference).
- Эта задача алгоритмически разрешима, и соответствующий алгоритм реализован в GHC.
- Таким образом, во многих случаях можно не указывать типы (программировать в бестиповом стиле), при этом сохраняя строгую типизацию.

Выведение типов

- Задача поиска наиболее общего типа (или выяснения, что терм нетипизуем) называется задачей *выведения типа* (type inference).
- Эта задача алгоритмически разрешима, и соответствующий алгоритм реализован в GHC.
- Таким образом, во многих случаях можно не указывать типы (программировать в бестиповом стиле), при этом сохраняя строгую типизацию.
- Однако Haskell позволяет и указывать типы явно. Таким образом можно сделать тип менее общим: например, для `idfunc = (\x -> x) :: ((a -> a) -> (a -> a))` применение `idfunc 0` будет некорректным.

Выведение типов

- Задача поиска наиболее общего типа (или выяснения, что терм нетипизуем) называется задачей *выведения типа* (type inference).
- Эта задача алгоритмически разрешима, и соответствующий алгоритм реализован в GHC.
- Таким образом, во многих случаях можно не указывать типы (программировать в бестиповом стиле), при этом сохраняя строгую типизацию.
- Однако Haskell позволяет и указывать типы явно. Таким образом можно сделать тип менее общим: например, для `idfunc = (\x -> x) :: ((a -> a) -> (a -> a))` применение `idfunc 0` будет некорректным.
- Также явное указание типов делает код яснее.

Выведение типов

- Сложность задачи выведения типов в общем случае, к сожалению, экспоненциальная.

Выведение типов

- Сложность задачи выведения типов в общем случае, к сожалению, экспоненциальная.
- Это неизбежно, потому что ответ может иметь экспоненциальную длину.

Выведение типов

- Сложность задачи выведения типов в общем случае, к сожалению, экспоненциальная.
- Это неизбежно, потому что ответ может иметь экспоненциальную длину.
  - Пример: для `dup = \x -> (x,x)` терм `dup . dup . dup ...` будет иметь экспоненциально длинный тип:

```
(dup . dup . dup . dup) ::
     b -> ((((b, b), (b, b)), ((b, b), (b, b))),
          (((b, b), (b, b)), ((b, b), (b, b))))
```

Выведение типов

- Сложность задачи выведения типов в общем случае, к сожалению, экспоненциальная.
- Это неизбежно, потому что ответ может иметь экспоненциальную длину.
    - Пример: для `dup = \x -> (x,x)` терм `dup . dup . dup ...` будет иметь экспоненциально длинный тип:

      ```
      (dup . dup . dup . dup) ::
           b -> ((((b, b), (b, b)), ((b, b), (b, b))),
                 (((b, b), (b, b)), ((b, b), (b, b))))
      ```
    - Здесь `.` — это инфиксно записанный **B**-комбинатор (композиция).

Выведение типов

- Сложность задачи выведения типов в общем случае, к сожалению, экспоненциальная.
- Это неизбежно, потому что ответ может иметь экспоненциальную длину.
    - Пример: для `dup = \x -> (x,x)` терм `dup . dup . dup ...` будет иметь экспоненциально длинный тип:

      ```
      (dup . dup . dup . dup) ::
           b -> ((((b, b), (b, b)), ((b, b), (b, b))),
                (((b, b), (b, b)), ((b, b), (b, b))))
      ```
    - Здесь `.` — это инфиксно записанный **B**-комбинатор (композиция).
    - Можно сделать то же и в чистой λ'е:
      `dup' = \x -> \f -> (f x x)`

Далее...

- На следующей лекции мы опишем алгоритм выведения типов в $\lambda_{\to}$ (простая система типов по Карри).

Далее...

- На следующей лекции мы опишем алгоритм выведения типов в $\lambda_{\to}$ (простая система типов по Карри).
- В частности, мы докажем теорему, что любой типизуемый терм имеет наиболее общий тип.

Далее...

- На следующей лекции мы опишем алгоритм выведения типов в $\lambda_{\to}$ (простая система типов по Карри).
- В частности, мы докажем теорему, что любой типизуемый терм имеет наиболее общий тип.
- После мы рассмотрим более богатые системы типов, такие как $\lambda 2$ (система F) и система Хиндли – Милнера, и обсудим вопросы выведения типов в этих системах.

Полиморфная типизация по Карри

- При типизации по Карри утверждения о типизации $\Gamma \vdash u : B$ *доказываются* в исчислении

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathrm{Ax} \qquad \frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash (\lambda x.u) : (A \to B)}\ \mathrm{Abs}$$

$$\frac{\Gamma \vdash u : (A \to B) \quad \Gamma \vdash v : A}{\Gamma \vdash (uv) : B}\ \mathrm{App}$$

Полиморфная типизация по Карри

- При типизации по Карри утверждения о типизации $\Gamma \vdash u : B$ *доказываются* в исчислении

$$\frac{}{\Gamma, x : A \vdash x : A}\ \text{Ax} \qquad \frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash (\lambda x.u) : (A \to B)}\ \text{Abs}$$

$$\frac{\Gamma \vdash u : (A \to B) \quad \Gamma \vdash v : A}{\Gamma \vdash (uv) : B}\ \text{App}$$

- Правила вывода соответствуют правилам построения термов.

Полиморфная типизация по Карри

- При типизации по Карри утверждения о типизации $\Gamma \vdash u : B$ *доказываются* в исчислении

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathrm{Ax} \qquad \frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash (\lambda x.u) : (A \to B)}\ \mathrm{Abs}$$

$$\frac{\Gamma \vdash u : (A \to B) \quad \Gamma \vdash v : A}{\Gamma \vdash (uv) : B}\ \mathrm{App}$$

- Правила вывода соответствуют правилам построения термов.
- Можно доказать, что доказуемость $\Gamma \vdash u : B$ равносильна возможности указать типы связанных переменных так, чтобы u получил тип B (типизация по Чёрчу).

Свойства типизации

- *Безопасность типов.* Если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.

Свойства типизации

- *Безопасность типов.* Если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.
- Отсюда следует *статичность* типизации: типы достаточно проверить один раз до начала вычислений.

Свойства типизации

- *Безопасность типов.* Если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.
- Отсюда следует *статичность* типизации: типы достаточно проверить один раз до начала вычислений.
- Безопасность типов доказывается индукцией по построению терма u.

Свойства типизации

- *Безопасность типов.* Если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.
- Отсюда следует *статичность* типизации: типы достаточно проверить один раз до начала вычислений.
- Безопасность типов доказывается индукцией по построению терма u.
    - Основная лемма: если $\Gamma \vdash u : B$, $\Gamma \vdash v : A$ и $(x : A) \in \Gamma$, то $\Gamma \vdash u[x := v] : B$.

Свойства типизации

- *Безопасность типов.* Если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.
- Отсюда следует *статичность* типизации: типы достаточно проверить один раз до начала вычислений.
- Безопасность типов доказывается индукцией по построению терма u.
    - Основная лемма: если $\Gamma \vdash u : B$, $\Gamma \vdash v : A$ и $(x : A) \in \Gamma$, то $\Gamma \vdash u[x := v] : B$.
- *Подстановка типов.* Утверждение о типизации сохраняет силу при подстановке сложных типов вместо переменных: если $\Gamma \vdash u : B$, то $\Gamma[p := A] \vdash u : B[p := A]$.

Свойства типизации

- *Безопасность типов.* Если $\Gamma \vdash u : B$ и $u \to_\beta u'$, то $\Gamma \vdash u' : B$.
- Отсюда следует *статичность* типизации: типы достаточно проверить один раз до начала вычислений.
- Безопасность типов доказывается индукцией по построению терма u.
    - Основная лемма: если $\Gamma \vdash u : B$, $\Gamma \vdash v : A$ и $(x : A) \in \Gamma$, то $\Gamma \vdash u[x := v] : B$.
- *Подстановка типов.* Утверждение о типизации сохраняет силу при подстановке сложных типов вместо переменных: если $\Gamma \vdash u : B$, то $\Gamma[p := A] \vdash u : B[p := A]$.
- При этом если переменная r не встречается в Γ, то из $\Gamma \vdash u : B$ следует $\Gamma \vdash u : B[r := A]$.

Задача выведения типов

- Наша задача, для данного контекста Γ_0 и данного терма u_0, определить, какие типы может иметь данный терм в данном контексте:

$$\Gamma_0 \vdash u_0 \ : \ ?$$

Задача выведения типов

- Наша задача, для данного контекста Γ_0 и данного терма u_0, определить, какие типы может иметь данный терм в данном контексте:
$$\Gamma_0 \vdash u_0 \ : \ ?$$
- Пусть $p_1, \ldots, p_n$ — базовые типы, входящие в контекст Γ_0. Будем считать их *неизменяемыми* (константами).

Задача выведения типов

- Наша задача, для данного контекста Γ_0 и данного терма u_0, определить, какие типы может иметь данный терм в данном контексте:
$$\Gamma_0 \vdash u_0 \; : \; ?$$
- Пусть $p_1, \ldots, p_n$ — базовые типы, входящие в контекст Γ_0. Будем считать их *неизменяемыми* (константами).
- Остальные базовые типы $r_1, r_2, \ldots$ — это настоящие *переменные*, вместо них можно будет подставлять другие типы.

Задача выведения типов

- Наша задача, для данного контекста Γ_0 и данного терма u_0, определить, какие типы может иметь данный терм в данном контексте:

$$\Gamma_0 \vdash u_0 \; : \; ?$$

- Пусть $p_1, \ldots, p_n$ — базовые типы, входящие в контекст Γ_0. Будем считать их *неизменяемыми* (константами).
- Остальные базовые типы $r_1, r_2, \ldots$ — это настоящие *переменные,* вместо них можно будет подставлять другие типы.
- Заметим, что в процессе вывода в контексте Γ также могут появиться переменные r_i, за счёт применения правила Abs.

Задача выведения типов

- Легко видеть, что если $\sigma = [r_1 := A_1, \ldots, r_k := A_k]$ — подстановка и $\Gamma_0 \vdash u_0 : B$, то $\Gamma_0 \vdash u_0 : B\sigma$.

- Легко видеть, что если $\sigma = [r_1 := A_1, \dots, r_k := A_k]$ — подстановка и $\Gamma_0 \vdash u_0 : B$, то $\Gamma_0 \vdash u_0 : B\sigma$.
  - Через $B\sigma$ будем обозначать применение подстановки σ к типу B.

Задача выведения типов

- Легко видеть, что если $\sigma = [r_1 := A_1, \dots, r_k := A_k]$ — подстановка и $\Gamma_0 \vdash u_0 : B$, то $\Gamma_0 \vdash u_0 : B\sigma$.
    - Через $B\sigma$ будем обозначать применение подстановки σ к типу B.
    - Тип B называется *более общим типом*, чем $B\sigma$, а $B\sigma$ — *более конкретным*, чем B.

Задача выведения типов

- Легко видеть, что если $\sigma = [r_1 := A_1, \ldots, r_k := A_k]$ — подстановка и $\Gamma_0 \vdash u_0 : B$, то $\Gamma_0 \vdash u_0 : B\sigma$.
    - Через $B\sigma$ будем обозначать применение подстановки σ к типу B.
    - Тип B называется *более общим типом*, чем $B\sigma$, а $B\sigma$ — *более конкретным*, чем B.
- Тип B_0 называется *наиболее общим типом* для u_0 в контексте Γ_0, если: (1) $\Gamma_0 \vdash u_0 : B_0$; (2) если $\Gamma_0 \vdash u_0 : B$, то $B = B_0\sigma$ для некоторой подстановки σ.

Задача выведения типов

Теорема

Для любых Γ_0 *и* u_0 *либо существует наиболее общий тип для* u_0 *в контексте* Γ_0, *либо* u_0 *не типизуем в контексте* Γ_0.

Задача выведения типов

Теорема

Для любых Γ_0 *и* u_0 *либо существует наиболее общий тип для* u_0 *в контексте* Γ_0*, либо* u_0 *не типизуем в контексте* Γ_0*.*

- Таким образом, задача выведения типов — это задача поиска наиболее общего типа (или определения, что терм не типизуем).

Задача выведения типов

Теорема

Для любых Γ_0 *и* u_0 *либо существует наиболее общий тип для* u_0 *в контексте* Γ_0, *либо* u_0 *не типизуем в контексте* Γ_0.

- Таким образом, задача выведения типов — это задача поиска наиболее общего типа (или определения, что терм не типизуем).
- Мы предъявим алгоритм решения этой задачи (тем самым, в частности, доказав теорему).

Задача выведения типов

Теорема

Для любых Γ_0 *и* u_0 *либо существует наиболее общий тип для* u_0 *в контексте* Γ_0*, либо* u_0 *не типизуем в контексте* Γ_0*.*

- Таким образом, задача выведения типов — это задача поиска наиболее общего типа (или определения, что терм не типизуем).
- Мы предъявим алгоритм решения этой задачи (тем самым, в частности, доказав теорему).
  - Отметим, что задача выведения типов проще, чем обычные задачи доказуемости, поскольку структура дерева вывода (последовательность применения правил) уже задана структурой терма.

«Желательные равенства»

- Начнём с того, что напишем вместо всех типов в правилах вывода различные переменные:

$$\frac{}{\Gamma, x : A \vdash x : r}\ \text{Ax}$$

$$\frac{\Gamma \vdash u : r_2 \quad \Gamma \vdash v : r_3}{\Gamma \vdash (uv) : r_1}\ \text{App}$$

$$\frac{\Gamma, x : r_2 \vdash u : r_3}{\Gamma \vdash (\lambda x.u) : r_1}\ \text{Abs}$$

«Желательные равенства»

- Начнём с того, что напишем вместо всех типов в правилах вывода различные переменные:

$$\frac{}{\Gamma, x : A \vdash x : r}\ \text{Ax} \qquad\qquad r \approx A$$

$$\frac{\Gamma \vdash u : r_2 \quad \Gamma \vdash v : r_3}{\Gamma \vdash (uv) : r_1}\ \text{App} \qquad\qquad r_2 \approx r_3 \to r_1$$

$$\frac{\Gamma, x : r_2 \vdash u : r_3}{\Gamma \vdash (\lambda x.u) : r_1}\ \text{Abs} \qquad\qquad r_1 \approx r_2 \to r_3$$

- Разумеется, здесь типы рассогласованы, и чтобы согласовать их, мы выпишем «желательные равенства».

«Желательные равенства»

- Начнём с того, что напишем вместо всех типов в правилах вывода различные переменные:

$$\frac{}{\Gamma, x : A \vdash x : r}\ \text{Ax} \qquad\qquad r \approx A$$

$$\frac{\Gamma \vdash u : r_2 \quad \Gamma \vdash v : r_3}{\Gamma \vdash (uv) : r_1}\ \text{App} \qquad\qquad r_2 \approx r_3 \to r_1$$

$$\frac{\Gamma, x : r_2 \vdash u : r_3}{\Gamma \vdash (\lambda x.u) : r_1}\ \text{Abs} \qquad\qquad r_1 \approx r_2 \to r_3$$

- Разумеется, здесь типы рассогласованы, и чтобы согласовать их, мы выпишем «желательные равенства».
- Если подстановка τ обращает все «желательные равенства» в равенства, то $r_0\tau$ — корректный тип для u_0 в контексте Γ_0.

Пример: комбинатор C

$$\mathbf{C} = \lambda f.\lambda x.\lambda y.f\,y\,x$$

Красный кардинал (Cardinalis cardinalis)

Пример: комбинатор C

$\mathbf{C} = \lambda f.\lambda x.\lambda y.f\,y\,x$

Пример: комбинатор C

$\mathbf{C} = \lambda f.\lambda x.\lambda y.f y x$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_9 \quad y : r_5 \vdash y : r_{10}}{f : r_1, y : r_5 \vdash f y : r_7}\,\text{App} \qquad x : r_3 \vdash x : r_8}{f : r_1, x : r_3, y : r_5 \vdash f y x : r_6}\,\text{App}}{f : r_1, x : r_3 \vdash \lambda y.f y x : r_4}\,\text{Abs}}{f : r_1 \vdash \lambda x.\lambda y.f y x : r_2}\,\text{Abs}}{\vdash \lambda f.\lambda x.\lambda y.f y x : r_0}\,\text{Abs}
$$

Пример: комбинатор C

$\mathbf{C} = \lambda f.\lambda x.\lambda y.fyx$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_9 \quad y : r_5 \vdash y : r_{10}}{f : r_1, y : r_5 \vdash fy : r_7}\,\mathrm{App} \qquad x : r_3 \vdash x : r_8}{f : r_1, x : r_3, y : r_5 \vdash fyx : r_6}\,\mathrm{App}}{f : r_1, x : r_3 \vdash \lambda y.fyx : r_4}\,\mathrm{Abs}}{f : r_1 \vdash \lambda x.\lambda y.fyx : r_2}\,\mathrm{Abs}}{\vdash \lambda f.\lambda x.\lambda y.fyx : r_0}\,\mathrm{Abs}
$$

«Желательные равенства»:

$$
\begin{array}{lll}
r_{10} \approx r_5 & r_9 \approx r_{10} \to r_7 & r_4 \approx r_5 \to r_6 \\
r_9 \approx r_1 & r_7 \approx r_8 \to r_6 & r_2 \approx r_3 \to r_4 \\
r_8 \approx r_3 & & r_0 \approx r_1 \to r_2
\end{array}
$$

Пример: комбинатор C

$\mathbf{C} = \lambda f.\lambda x.\lambda y.fyx$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_9 \qquad y : r_5 \vdash y : r_{10}}{f : r_1, y : r_5 \vdash fy : r_7}\,\text{App} \qquad x : r_3 \vdash x : r_8}{f : r_1, x : r_3, y : r_5 \vdash fyx : r_6}\,\text{App}}{f : r_1, x : r_3 \vdash \lambda y.fyx : r_4}\,\text{Abs}}{f : r_1 \vdash \lambda x.\lambda y.fyx : r_2}\,\text{Abs}}{\vdash \lambda f.\lambda x.\lambda y.fyx : r_0}\,\text{Abs}
$$

«Желательные равенства»:

$r_{10} := r_5$	$r_1 \approx r_5 \to r_7$	$r_4 \approx r_5 \to r_6$
$r_9 := r_1$	$r_7 \approx r_3 \to r_6$	$r_2 \approx r_3 \to r_4$
$r_8 := r_3$		$r_0 \approx r_1 \to r_2$

Пример: комбинатор C

$\mathbf{C} = \lambda f.\lambda x.\lambda y.f y x$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_9 \qquad y : r_5 \vdash y : r_{10}}{f : r_1, y : r_5 \vdash f y : r_7}\,\text{App} \qquad x : r_3 \vdash x : r_8}{f : r_1, x : r_3, y : r_5 \vdash f y x : r_6}\,\text{App}}{f : r_1, x : r_3 \vdash \lambda y.f y x : r_4}\,\text{Abs}}{f : r_1 \vdash \lambda x.\lambda y.f y x : r_2}\,\text{Abs}}{\vdash \lambda f.\lambda x.\lambda y.f y x : r_0}\,\text{Abs}
$$

«Желательные равенства»:

$$
\begin{array}{lll}
r_{10} := r_5 & r_1 := r_5 \to r_7 & r_4 \approx r_5 \to r_6 \\
r_9 := r_1 & r_7 \approx {\color{blue} r_3} \to r_6 & r_2 \approx r_3 \to r_4 \\
r_8 := r_3 & & r_0 \approx {\color{blue}(r_5 \to r_7)} \to r_2
\end{array}
$$

Пример: комбинатор C

$\mathbf{C} = \lambda f.\lambda x.\lambda y.fyx$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_9 \quad y : r_5 \vdash y : r_{10}}{f : r_1, y : r_5 \vdash fy : r_7}\,\text{App} \qquad x : r_3 \vdash x : r_8}{f : r_1, x : r_3, y : r_5 \vdash fyx : r_6}\,\text{App}}{f : r_1, x : r_3 \vdash \lambda y.fyx : r_4}\,\text{Abs}}{f : r_1 \vdash \lambda x.\lambda y.fyx : r_2}\,\text{Abs}}{\vdash \lambda f.\lambda x.\lambda y.fyx : r_0}\,\text{Abs}}{}$$

«Желательные равенства»:

$$\begin{array}{lll}
r_{10} := r_5 & r_1 := r_5 \to r_7 & r_4 \approx r_5 \to r_6 \\
r_9 := r_1 & r_7 := r_3 \to r_6 & r_2 \approx r_3 \to r_4 \\
r_8 := r_3 & & r_0 \approx (r_5 \to (r_3 \to r_6)) \to r_2
\end{array}$$

Пример: комбинатор C

$\mathbf{C} = \lambda f.\lambda x.\lambda y.f y x$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_9 \quad y : r_5 \vdash y : r_{10}}{f : r_1, y : r_5 \vdash f y : r_7}\,\text{App} \qquad x : r_3 \vdash x : r_8}{f : r_1, x : r_3, y : r_5 \vdash f y x : r_6}\,\text{App}}{f : r_1, x : r_3 \vdash \lambda y.f y x : r_4}\,\text{Abs}}{f : r_1 \vdash \lambda x.\lambda y.f y x : r_2}\,\text{Abs}}{\vdash \lambda f.\lambda x.\lambda y.f y x : r_0}\,\text{Abs}
$$

«Желательные равенства»:

$$
\begin{array}{lll}
r_{10} := r_5 & r_1 := r_5 \to r_7 & r_4 := r_5 \to r_6 \\
r_9 := r_1 & r_7 := r_3 \to r_6 & r_2 \approx r_3 \to (r_5 \to r_6) \\
r_8 := r_3 & & r_0 \approx (r_5 \to (r_3 \to r_6)) \to r_2
\end{array}
$$

Пример: комбинатор C

$\mathbf{C} = \lambda f.\lambda x.\lambda y.fyx$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{f : r_1 \vdash f : r_9 \quad y : r_5 \vdash y : r_{10}}{f : r_1, y : r_5 \vdash fy : r_7}\text{App} \qquad x : r_3 \vdash x : r_8}{f : r_1, x : r_3, y : r_5 \vdash fyx : r_6}\text{App}}{f : r_1, x : r_3 \vdash \lambda y.fyx : r_4}\text{Abs}}{f : r_1 \vdash \lambda x.\lambda y.fyx : r_2}\text{Abs}}{\vdash \lambda f.\lambda x.\lambda y.fyx : r_0}\text{Abs}$$

«Желательные равенства»:

$$\begin{array}{lll} r_{10} := r_5 & r_1 := r_5 \to r_7 & r_4 := r_5 \to r_6 \\ r_9 := r_1 & r_7 := r_3 \to r_6 & r_2 := r_3 \to (r_5 \to r_6) \\ r_8 := r_3 & & r_0 \approx (r_5 \to (r_3 \to r_6)) \to (r_3 \to (r_5 \to r_6)) \end{array}$$

Пример: комбинатор C

$\mathbf{C} = \lambda f.\lambda x.\lambda y.fyx$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_9 \quad y : r_5 \vdash y : r_{10}}{f : r_1, y : r_5 \vdash fy : r_7}\ \text{App} \qquad x : r_3 \vdash x : r_8}{f : r_1, x : r_3, y : r_5 \vdash fyx : r_6}\ \text{App}}{f : r_1, x : r_3 \vdash \lambda y.fyx : r_4}\ \text{Abs}}{f : r_1 \vdash \lambda x.\lambda y.fyx : r_2}\ \text{Abs}}{\vdash \lambda f.\lambda x.\lambda y.fyx : r_0}\ \text{Abs}$$

«Желательные равенства»:

$$\begin{array}{lll}
r_{10} := r_5 & r_1 := r_5 \to r_7 & r_4 := r_5 \to r_6 \\
r_9 := r_1 & r_7 := r_3 \to r_6 & r_2 := r_3 \to (r_5 \to r_6) \\
r_8 := r_3 & & r_0 := (r_5 \to (r_3 \to r_6)) \to (r_3 \to (r_5 \to r_6))
\end{array}$$

Пример: комбинатор C

$$
\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{f : r_5 \to (r_3 \to r_6) \vdash f : r_5 \to (r_3 \to r_6) \qquad y : r_5 \vdash y : r_5}{f : r_5 \to (r_3 \to r_6), y : r_5 \vdash f y : r_3 \to r_6}\ \text{App} \qquad x : r_3 \vdash x : r_3}{f : r_5 \to (r_3 \to r_6), x : r_3, y : r_5 \vdash f y x : r_6}\ \text{App}}{f : r_5 \to (r_3 \to r_6), x : r_3 \vdash \lambda y. f y x : (r_5 \to r_6)}\ \text{Abs}}{f : r_5 \to (r_3 \to r_6) \vdash \lambda x. \lambda y. f y x : r_3 \to (r_5 \to r_6)}\ \text{Abs}}{\vdash \lambda f. \lambda x. \lambda y. f y x : (r_5 \to (r_3 \to r_6)) \to (r_3 \to (r_5 \to r_6))}\ \text{Abs}
$$

Пример: комбинатор C

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{f : r_5 \to (r_3 \to r_6) \vdash f : r_5 \to (r_3 \to r_6) \quad y : r_5 \vdash y : r_5}{f : r_5 \to (r_3 \to r_6), y : r_5 \vdash f y : r_3 \to r_6}\text{App} \quad x : r_3 \vdash x : r_3}{f : r_5 \to (r_3 \to r_6), x : r_3, y : r_5 \vdash f y x : r_6}\text{App}}{f : r_5 \to (r_3 \to r_6), x : r_3 \vdash \lambda y. f y x : (r_5 \to r_6)}\text{Abs}}{f : r_5 \to (r_3 \to r_6) \vdash \lambda x.\lambda y. f y x : r_3 \to (r_5 \to r_6)}\text{Abs}}{\vdash \lambda f.\lambda x.\lambda y. f y x : (r_5 \to (r_3 \to r_6)) \to (r_3 \to (r_5 \to r_6))}\text{Abs}$$

Вместо r_5, r_3, r_6 можно подставить произвольные типы:
$\mathbf{C} : (A \to (B \to C)) \to (B \to (A \to C))$.

Пример: комбинатор Y

$$\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$$

Пример: комбинатор Y

$$\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$$

$$
\dfrac{
  \dfrac{
    \dfrac{
      f : r_1 \vdash f : r_7
      \qquad
      \dfrac{x : r_5 \vdash x : r_9 \qquad x : r_5 \vdash x : r_{10}}{x : r_5 \vdash xx : r_8}\ \text{App}
    }{
      f : r_1, x : r_5 \vdash f(xx) : r_6
    }\ \text{App}
  }{
    f : r_1 \vdash \lambda x.f(xx) : r_3
  }\ \text{Abs}
  \qquad
  \dfrac{\dots}{f : r_1 \vdash \lambda x.f(xx) : r_4}
}{
  \dfrac{f : r_1 \vdash (\lambda x.f(xx))(\lambda x.f(xx)) : r_2}{\vdash \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big) : r_0}\ \text{Abs}
}\ \text{App}
$$

Пример: комбинатор Y

$$\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_7 \qquad \dfrac{x : r_5 \vdash x : r_9 \qquad x : r_5 \vdash x : r_{10}}{x : r_5 \vdash xx : r_8}\ \mathrm{App}}{f : r_1, x : r_5 \vdash f(xx) : r_6}\ \mathrm{App}}{f : r_1 \vdash \lambda x.f(xx) : r_3}\ \mathrm{Abs} \qquad \dfrac{\dots}{f : r_1 \vdash \lambda x.f(xx) : r_4}}{f : r_1 \vdash (\lambda x.f(xx))(\lambda x.f(xx)) : r_2}\ \mathrm{App}}{\vdash \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big) : r_0}\ \mathrm{Abs}$$

Пример: комбинатор Y

$$\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$$

$$\dfrac{f : r_1 \vdash f : r_7 \qquad \dfrac{x : r_5 \vdash x : r_9 \quad x : r_5 \vdash x : r_{10}}{x : r_5 \vdash xx : r_8}\ \text{App}}{\dfrac{f : r_1, x : r_5 \vdash f(xx) : r_6}{\dfrac{f : r_1 \vdash \lambda x.f(xx) : r_3 \qquad \dfrac{\dots}{f : r_1 \vdash \lambda x.f(xx) : r_4}}{\dfrac{f : r_1 \vdash (\lambda x.f(xx))(\lambda x.f(xx)) : r_2}{\vdash \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big) : r_0}\ \text{Abs}}\ \text{App}}\ \text{Abs}}\ \text{App}$$

$r_{10} \approx r_5$

$r_9 \approx r_5$

$r_9 \approx r_{10} \to r_8$

…

Пример: комбинатор Y

$$\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$$

$$\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_7 \quad \dfrac{x : r_5 \vdash x : r_9 \quad x : r_5 \vdash x : r_{10}}{x : r_5 \vdash xx : r_8}\ \text{App}}{f : r_1, x : r_5 \vdash f(xx) : r_6}\ \text{App}}{f : r_1 \vdash \lambda x.f(xx) : r_3}\ \text{Abs} \qquad \dfrac{\dots}{f : r_1 \vdash \lambda x.f(xx) : r_4}}{\dfrac{f : r_1 \vdash (\lambda x.f(xx))(\lambda x.f(xx)) : r_2}{\vdash \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big) : r_0}\ \text{Abs}}\ \text{App}$$

$r_{10} := r_5$

$r_9 := r_5$

$r_5 \approx r_5 \to r_8$

…

Пример: комбинатор Y

$$\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$$

$$\dfrac{\dfrac{\dfrac{f : r_1 \vdash f : r_7 \qquad \dfrac{x : r_5 \vdash x : r_9 \qquad x : r_5 \vdash x : r_{10}}{x : r_5 \vdash xx : r_8}\,\text{App}}{f : r_1, x : r_5 \vdash f(xx) : r_6}\,\text{App}}{f : r_1 \vdash \lambda x.f(xx) : r_3}\,\text{Abs} \qquad \dfrac{\dots}{f : r_1 \vdash \lambda x.f(xx) : r_4}}{\dfrac{f : r_1 \vdash (\lambda x.f(xx))(\lambda x.f(xx)) : r_2}{\vdash \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big) : r_0}\,\text{Abs}}\,\text{App}$$

$r_{10} := r_5$

$r_9 := r_5$

$r_5 \approx r_5 \to r_8$

…

Унификация

- В общем случае задача выведения типа сводится к *задаче унификации системы «желательных равенств».*

Унификация

- В общем случае задача выведения типа сводится к *задаче унификации системы «желательных равенств».*
- Подстановка τ *унифицирует* систему равенств $\mathcal{S} = \{A_1 \approx B_1, \ldots, A_n \approx B_n\}$, если $A_i\tau = B_i\tau$ для каждого i.

Унификация

- В общем случае задача выведения типа сводится к *задаче унификации системы «желательных равенств»*.
- Подстановка τ *унифицирует* систему равенств $\mathcal{S} = \{A_1 \approx B_1, \dots, A_n \approx B_n\}$, если $A_i\tau = B_i\tau$ для каждого i.
- τ_0 — наиболее общий унификатор (most general unifier, MGU) системы $\mathcal{S}$, если (1) τ_0 унифицирует $\mathcal{S}$; (2) для любого другого унификатора τ существует подстановка σ, такая что $\tau = \tau_0 \circ \sigma$.

Унификация

- В общем случае задача выведения типа сводится к *задаче унификации системы «желательных равенств».*
- Подстановка τ *унифицирует* систему равенств $\mathcal{S} = \{A_1 \approx B_1, \ldots, A_n \approx B_n\}$, если $A_i\tau = B_i\tau$ для каждого i.
- τ_0 — наиболее общий унификатор (most general unifier, MGU) системы $\mathcal{S}$, если (1) τ_0 унифицирует $\mathcal{S}$; (2) для любого другого унификатора τ существует подстановка σ, такая что $\tau = \tau_0 \circ \sigma$.
    - Здесь сначала применяется τ_0, потом σ.

Унификация

- В общем случае задача выведения типа сводится к *задаче унификации системы «желательных равенств»*.
- Подстановка τ *унифицирует* систему равенств $\mathcal{S} = \{A_1 \approx B_1, \dots, A_n \approx B_n\}$, если $A_i\tau = B_i\tau$ для каждого i.
- τ_0 — наиболее общий унификатор (most general unifier, MGU) системы $\mathcal{S}$, если (1) τ_0 унифицирует $\mathcal{S}$; (2) для любого другого унификатора τ существует подстановка σ, такая что $\tau = \tau_0 \circ \sigma$.
    - Здесь сначала применяется τ_0, потом σ.
    - Таким образом, τ является «конкретизацией» τ_0.

Унификация и типизация

- Обозначим через $\mathscr{S}_{\Gamma,u}$ систему «желательных равенств», возникающую при типизации терма u в контексте Γ.

Унификация и типизация

- Обозначим через $\mathcal{S}_{\Gamma,u}$ систему «желательных равенств», возникающую при типизации терма u в контексте Γ.
- Всякий унификатор τ этой системы даёт типизацию u в Γ, а именно $B = r_0\tau$.

Унификация и типизация

- Обозначим через $\mathcal{S}_{\Gamma,u}$ систему «желательных равенств», возникающую при типизации терма u в контексте Γ.
- Всякий унификатор τ этой системы даёт типизацию u в Γ, а именно $B = r_0\tau$.
- И наоборот, если $\Gamma \vdash u : B$, то из доказательства этого утверждения извлекается подстановка τ, унифицирующая $\mathcal{S}_{\Gamma,u}$; при этом $B = r_0\tau$.

Унификация и типизация

- Обозначим через $\mathcal{S}_{\Gamma,u}$ систему «желательных равенств», возникающую при типизации терма u в контексте Γ.
- Всякий унификатор τ этой системы даёт типизацию u в Γ, а именно $B = r_0\tau$.
- И наоборот, если $\Gamma \vdash u : B$, то из доказательства этого утверждения извлекается подстановка τ, унифицирующая $\mathcal{S}_{\Gamma,u}$; при этом $B = r_0\tau$.
- MGU τ_0 даёт наиболее общий тип $B_0 = r_0\tau_0$. Действительно, любой другой B имеет вид $r_0\tau = r_0\tau_0\sigma = B_0\sigma$ (т.к. $\tau = \tau_0 \circ \sigma$).

Унификация и типизация

- Обозначим через $\mathcal{S}_{\Gamma,u}$ систему «желательных равенств», возникающую при типизации терма u в контексте Γ.
- Всякий унификатор τ этой системы даёт типизацию u в Γ, а именно $B = r_0\tau$.
- И наоборот, если $\Gamma \vdash u : B$, то из доказательства этого утверждения извлекается подстановка τ, унифицирующая $\mathcal{S}_{\Gamma,u}$; при этом $B = r_0\tau$.
- MGU τ_0 даёт наиболее общий тип $B_0 = r_0\tau_0$. Действительно, любой другой B имеет вид $r_0\tau = r_0\tau_0\sigma = B_0\sigma$ (т.к. $\tau = \tau_0 \circ \sigma$).
- Значит, достаточно найти MGU — или установить, что унификаторов нет вообще, и тогда терм нетипизуем.

Алгоритм Робинсона

- Алгоритм Робинсона находит MGU данной системы $\mathcal{S}$ или устанавливает, что система неунифицируема.

Алгоритм Робинсона

- Алгоритм Робинсона находит MGU данной системы $\mathcal{S}$ или устанавливает, что система неунифицируема.
- Алгоритм действует по шагам, упрощая $\mathcal{S}$; на каждом шаге множество унификаторов системы либо сохраняется, либо изменяется по известному правилу.

- Алгоритм Робинсона находит MGU данной системы $\mathscr{S}$ или устанавливает, что система неунифицируема.
- Алгоритм действует по шагам, упрощая $\mathscr{S}$; на каждом шаге множество унификаторов системы либо сохраняется, либо изменяется по известному правилу.
- При этом уменьшаются определённые параметры рекурсии, т.е. алгоритм завершает работу за конечное число шагов.

Алгоритм Робинсона

Возьмём произвольное (например, первое) равенство из системы $\mathcal{S}$ и рассмотрим возможные случаи.

Алгоритм Робинсона

Возьмём произвольное (например, первое) равенство из системы $\mathscr{S}$ и рассмотрим возможные случаи.

1. $A \approx A$. Такое равенство можно удалить без изменения множества унификаторов.

Алгоритм Робинсона

Возьмём произвольное (например, первое) равенство из системы $\mathcal{S}$ и рассмотрим возможные случаи.

1. $A \approx A$. Такое равенство можно удалить без изменения множества унификаторов.
2. $(A_1 \to A_2) \approx (B_1 \to B_2)$. Заменяем на два равенства: $A_1 \approx B_1$ и $A_2 \approx B_2$. Множество унификаторов не меняется.

Алгоритм Робинсона

Возьмём произвольное (например, первое) равенство из системы $\mathcal{S}$ и рассмотрим возможные случаи.

1. $A \approx A$. Такое равенство можно удалить без изменения множества унификаторов.
2. $(A_1 \to A_2) \approx (B_1 \to B_2)$. Заменяем на два равенства: $A_1 \approx B_1$ и $A_2 \approx B_2$. Множество унификаторов не меняется.
3. $p_i \approx B$ или $B \approx p_i$, где B — не переменная и не сама p_i. Такое равенство неунифицируемо, алгоритм останавливается.

Алгоритм Робинсона

Возьмём произвольное (например, первое) равенство из системы $\mathcal{S}$ и рассмотрим возможные случаи.

1. $A \approx A$. Такое равенство можно удалить без изменения множества унификаторов.
2. $(A_1 \to A_2) \approx (B_1 \to B_2)$. Заменяем на два равенства: $A_1 \approx B_1$ и $A_2 \approx B_2$. Множество унификаторов не меняется.
3. $p_i \approx B$ или $B \approx p_i$, где B — не переменная и не сама p_i. Такое равенство неунифицируемо, алгоритм останавливается.
4. $r_j \approx B$ или $B \approx r_j$, где B содержит r_j, но не совпадает с ним. Такое равенство тоже неунифицируемо.

Алгоритм Робинсона

5. Наиболее интересный случай: $r_j \approx A$ или $A \approx r_j$, где A не содержит r_j.

В этом случае нужна совершить подстановку $r_j \ := A$ и продолжить работать с системой без r_j.

Алгоритм Робинсона

5. Наиболее интересный случай: $r_j \approx A$ или $A \approx r_j$, где A не содержит r_j.

В этом случае нужна совершить подстановку $r_j := A$ и продолжить работать с системой без r_j.

- Запишем это более формально. Пусть $\mathcal{S}$ — исходная система, $\mathcal{S}'$ — система без равенства $r_j \approx A$.

5. Наиболее интересный случай: $r_j \approx A$ или $A \approx r_j$, где A не содержит r_j.

В этом случае нужна совершить подстановку $r_j := A$ и продолжить работать с системой без r_j.

- Запишем это более формально. Пусть $\mathcal{S}$ — исходная система, $\mathcal{S}'$ — система без равенства $r_j \approx A$.
- Утверждается, что τ — унификатор системы $\mathcal{S}$ тогда и только тогда, когда $\tau = [r_j := A] \circ \tau'$, где τ' унифицирует $\mathcal{S}'[r_j := A]$.

5. Наиболее интересный случай: $r_j \approx A$ или $A \approx r_j$, где A не содержит r_j.

В этом случае нужна совершить подстановку $r_j := A$ и продолжить работать с системой без r_j.

- Запишем это более формально. Пусть $\mathcal{S}$ — исходная система, $\mathcal{S}'$ — система без равенства $r_j \approx A$.
- Утверждается, что τ — унификатор системы $\mathcal{S}$ тогда и только тогда, когда $\tau = [r_j := A] \circ \tau'$, где τ' унифицирует $\mathcal{S}'[r_j := A]$.
- Таким образом, задача унификации $\mathcal{S}$ сводится к задаче унификации $\mathcal{S}'[r_j := A]$.

Алгоритм Робинсона

- Действительно, пусть τ' — унификатор системы $\mathcal{S}'[r_j := A]$, и пусть $\tau = [r_j := A] \circ \tau'$.

Алгоритм Робинсона

- Действительно, пусть τ' — унификатор системы $\mathcal{S}'[r_j := A]$, и пусть $\tau = [r_j := A] \circ \tau'$.
- Тогда после подстановки $r_j := A$ типы r_j и A отождествятся (поскольку A не содержит r_j).

Алгоритм Робинсона

- Действительно, пусть τ' — унификатор системы $\mathcal{S}'[r_j := A]$, и пусть $\tau = [r_j := A] \circ \tau'$.
- Тогда после подстановки $r_j := A$ типы r_j и A отождествятся (поскольку A не содержит r_j).
- Остальные равенства (после подстановки $r_j := A$) унифицирует подстановка τ'.

Алгоритм Робинсона

- Действительно, пусть τ' — унификатор системы $\mathcal{S}'[r_j := A]$, и пусть $\tau = [r_j := A] \circ \tau'$.
- Тогда после подстановки $r_j := A$ типы r_j и A отождествятся (поскольку A не содержит r_j).
- Остальные равенства (после подстановки $r_j := A$) унифицирует подстановка τ'.
- Теперь пусть τ — унификатор $\mathcal{S}$.

Алгоритм Робинсона

- Действительно, пусть τ' — унификатор системы $\mathcal{S}'[r_j := A]$, и пусть $\tau = [r_j := A] \circ \tau'$.
- Тогда после подстановки $r_j := A$ типы r_j и A отождествятся (поскольку A не содержит r_j).
- Остальные равенства (после подстановки $r_j := A$) унифицирует подстановка τ'.
- Теперь пусть τ — унификатор $\mathcal{S}$.
- Значит, $r_j\tau = A\tau$.

Алгоритм Робинсона

- Действительно, пусть τ' — унификатор системы $\mathcal{S}'[r_j := A]$, и пусть $\tau = [r_j := A] \circ \tau'$.
- Тогда после подстановки $r_j := A$ типы r_j и A отождествятся (поскольку A не содержит r_j).
- Остальные равенства (после подстановки $r_j := A$) унифицирует подстановка τ'.
- Теперь пусть τ — унификатор $\mathcal{S}$.
- Значит, $r_j\tau = A\tau$.
- Для каждой r_i, где $i \neq j$, положим $r_i\tau' = r_i\tau$. (Подстановку достаточно определить на переменных.)

Алгоритм Робинсона

- Действительно, пусть τ' — унификатор системы $\mathcal{S}'[r_j := A]$, и пусть $\tau = [r_j := A] \circ \tau'$.
- Тогда после подстановки $r_j := A$ типы r_j и A отождествятся (поскольку A не содержит r_j).
- Остальные равенства (после подстановки $r_j := A$) унифицирует подстановка τ'.
- Теперь пусть τ — унификатор $\mathcal{S}$.
- Значит, $r_j\tau = A\tau$.
- Для каждой r_i, где $i \neq j$, положим $r_i\tau' = r_i\tau$. (Подстановку достаточно определить на переменных.)
- Легко проверить, что $\tau = [r_j := A] \circ \tau'$.

Алгоритм Робинсона

- Действительно, пусть τ' — унификатор системы $\mathcal{S}'[r_j := A]$, и пусть $\tau = [r_j := A] \circ \tau'$.
- Тогда после подстановки $r_j := A$ типы r_j и A отождествятся (поскольку A не содержит r_j).
- Остальные равенства (после подстановки $r_j := A$) унифицирует подстановка τ'.
- Теперь пусть τ — унификатор $\mathcal{S}$.
- Значит, $r_j\tau = A\tau$.
- Для каждой r_i, где $i \neq j$, положим $r_i\tau' = r_i\tau$. (Подстановку достаточно определить на переменных.)
- Легко проверить, что $\tau = [r_j := A] \circ \tau'$.
- $A_i[r_j := A]\tau' = A_i\tau = B_i\tau = B_i[r_j := A]\tau'$.

Алгоритм Робинсона

- Более того, если τ_0' — MGU системы $\mathcal{S}'[r_j := A]$, то $\tau = [r_j := A] \circ \tau'$ — MGU системы $\mathcal{S}$.

- Более того, если τ_0' — MGU системы $\mathcal{S}'[r_j := A]$, то $\tau = [r_j := A] \circ \tau'$ — MGU системы $\mathcal{S}$.
- Действительно, для любого другого унификатора τ имеем $\tau = [r_j := A] \circ \tau' = [r_j := A] \circ \tau_0' \circ \sigma = \tau_0 \circ \sigma$.

Алгоритм Робинсона

- Более того, если τ_0' — MGU системы $\mathcal{S}'[r_j := A]$, то $\tau = [r_j := A] \circ \tau'$ — MGU системы $\mathcal{S}$.
- Действительно, для любого другого унификатора τ имеем $\tau = [r_j := A] \circ \tau' = [r_j := A] \circ \tau_0' \circ \sigma = \tau_0 \circ \sigma$.
- Если $\mathcal{S}'[r_j := A]$ неунифицируема, то такова и исходная $\mathcal{S}$.

Алгоритм Робинсона

- Более того, если τ_0' — MGU системы $\mathcal{S}'[r_j := A]$, то $\tau = [r_j := A] \circ \tau'$ — MGU системы $\mathcal{S}$.
- Действительно, для любого другого унификатора τ имеем $\tau = [r_j := A] \circ \tau' = [r_j := A] \circ \tau_0' \circ \sigma = \tau_0 \circ \sigma$.
- Если $\mathcal{S}'[r_j := A]$ неунифицируема, то такова и исходная $\mathcal{S}$.
- Таким образом, мы последовательно сводим вычисление MGU к более простым системам.

Алгоритм Робинсона

- Более того, если τ_0' — MGU системы $\mathcal{S}'[r_j := A]$, то $\tau = [r_j := A] \circ \tau'$ — MGU системы $\mathcal{S}$.
- Действительно, для любого другого унификатора τ имеем $\tau = [r_j := A] \circ \tau' = [r_j := A] \circ \tau_0' \circ \sigma = \tau_0 \circ \sigma$.
- Если $\mathcal{S}'[r_j := A]$ неунифицируема, то такова и исходная $\mathcal{S}$.
- Таким образом, мы последовательно сводим вычисление MGU к более простым системам.
- Осталось доказать, что процесс сойдётся за конечное число шагов к пустой системе (её MGU — тождественная подстановка).

Алгоритм Робинсона

Параметры рекурсии (в порядке убывания приоритета):

1. Количество переменных (r_j): убывает в случае 5.
2. Количество стрелок: убывает в случае 2. При этом 1-й параметр сохраняется.
3. Количество равенств: убывает в случае 1. При этом 1-й и 2-й параметры не растут.

Алгоритм Робинсона

Параметры рекурсии (в порядке убывания приоритета):

1. Количество переменных (r_j): убывает в случае 5.
2. Количество стрелок: убывает в случае 2. При этом 1-й параметр сохраняется.
3. Количество равенств: убывает в случае 1. При этом 1-й и 2-й параметры не растут.

В случаях 3 и 4 алгоритм сразу останавливается.

- Подведём итоги.

- Подведём итоги.
- При типизации по Карри у каждого терма (в данном контексте) есть наиболее общий тип, а все остальные типы получаются из него подстановкой (конкретизацией).

- Подведём итоги.
- При типизации по Карри у каждого терма (в данном контексте) есть наиболее общий тип, а все остальные типы получаются из него подстановкой (конкретизацией).
    - Это *параметрический полиморфизм* (параметры — переменные $r_1, r_2, \ldots$).

- Подведём итоги.
- При типизации по Карри у каждого терма (в данном контексте) есть наиболее общий тип, а все остальные типы получаются из него подстановкой (конкретизацией).
    - Это *параметрический полиморфизм* (параметры — переменные $r_1, r_2, \ldots$).
- Типы не портятся и не используются при редукциях.

- Подведём итоги.
- При типизации по Карри у каждого терма (в данном контексте) есть наиболее общий тип, а все остальные типы получаются из него подстановкой (конкретизацией).
  - Это *параметрический полиморфизм* (параметры — переменные $r_1, r_2, \ldots$).
- Типы не портятся и не используются при редукциях.
  - Это даёт *статичность* типизации.

- Подведём итоги.
- При типизации по Карри у каждого терма (в данном контексте) есть наиболее общий тип, а все остальные типы получаются из него подстановкой (конкретизацией).
    - Это *параметрический полиморфизм* (параметры — переменные $r_1, r_2, \ldots$).
- Типы не портятся и не используются при редукциях.
    - Это даёт *статичность* типизации.
    - В Haskell'е второе неверно: информация о типах используется при вычислении (ad hoc полиморфизм).

- Подведём итоги.
- При типизации по Карри у каждого терма (в данном контексте) есть наиболее общий тип, а все остальные типы получаются из него подстановкой (конкретизацией).
    - Это *параметрический полиморфизм* (параметры — переменные $r_1, r_2, \ldots$).
- Типы не портятся и не используются при редукциях.
    - Это даёт *статичность* типизации.
    - В Haskell'е второе неверно: информация о типах используется при вычислении (ad hoc полиморфизм).
- *Выведение типов.* Наиболее общий тип можно вычислить, его не нужно указывать явно.

Типизация и рекурсия

- Как мы видели, комбинатор $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ нетипизируем в системе $\lambda_{\to}$.

Типизация и рекурсия

- Как мы видели, комбинатор $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ нетипизируем в системе $\lambda_{\to}$.
- Более того: имеет место *теорема о нормализуемости* — любой типизируемый терм сильно нормализуем.

- Как мы видели, комбинатор $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ нетипизируем в системе $\lambda_{\to}$.
- Более того: имеет место *теорема о нормализуемости* — любой типизируемый терм сильно нормализуем.
- Следовательно, *никакой* комбинатор неподвижной точки не может быть типизируемым.

- Как мы видели, комбинатор $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$ нетипизируем в системе $\lambda_{\to}$.
- Более того: имеет место *теорема о нормализуемости* — любой типизируемый терм сильно нормализуем.
- Следовательно, *никакой* комбинатор неподвижной точки не может быть типизируемым.
- Однако можно добавить константу $\mathbb{Y}$ с редукцией $\mathbb{Y}u \to_{\delta} u(\mathbb{Y}u)$ и полиморфным типом $(r \to r) \to r$.

Типизация и рекурсия

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст Γ.

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст Γ.
- Значит, входящая в него переменная станет неизменяемой, $\mathbb{Y}_p : (p \to p) \to p$.

Типизация и рекурсия

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст Γ.
- Значит, входящая в него переменная станет неизменяемой, $\mathbb{Y}_p : (p \to p) \to p$.
- Это плохо: нам *«не хватает полиморфизма»*, чтобы типизовать $\mathbb{Y}$.

Типизация и рекурсия

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст Γ.
- Значит, входящая в него переменная станет неизменяемой, $\mathbb{Y}_p : (p \to p) \to p$.
- Это плохо: нам *«не хватает полиморфизма»*, чтобы типизовать $\mathbb{Y}$.
- Можно обойти эту проблему, использовав конструктор термов Y:

$$\frac{\Gamma, x : r_2 \vdash u : r_3}{\Gamma \vdash (\mathrm{Y}x.u) : r_1}\ \mathrm{Fix} \qquad r_1 \approx r_2 \approx r_3 \qquad \mathrm{Y}x.u \to_\delta u[x := \mathrm{Y}x.u]$$

Типизация и рекурсия

- Однако утверждение о типизации константы $\mathbb{Y}$ придётся поместить в контекст Γ.
- Значит, входящая в него переменная станет неизменяемой, $\mathbb{Y}_p : (p \to p) \to p$.
- Это плохо: нам *«не хватает полиморфизма»*, чтобы типизовать $\mathbb{Y}$.
- Можно обойти эту проблему, использовав конструктор термов Y:

$$\frac{\Gamma, x : r_2 \vdash u : r_3}{\Gamma \vdash (\mathrm{Y}x.u) : r_1}\ \mathrm{Fix} \qquad r_1 \approx r_2 \approx r_3 \qquad \mathrm{Y}x.u \to_\delta u[x := \mathrm{Y}x.u]$$

- Безопасность типов при δ-редукции соблюдается.

Квантор $\forall$ по типам

- Неявно в полиморфной типизации по Карри присутствуют *кванторы всеобщности* по переменным r_j:

$$f : (p \to p) \vdash \lambda g.\lambda z.f(gz) : \forall r.((r \to p) \to r \to p).$$

Квантор ∀ по типам

- Неявно в полиморфной типизации по Карри присутствуют *кванторы всеобщности* по переменным r_j:

$$f : (p \to p) \vdash \lambda g.\lambda z.f(gz) : \forall r.((r \to p) \to r \to p).$$

- По смыслу, $\mathbb{Y} : \forall r.((r \to r) \to r)$, и мы хотим поместить эту декларацию в контекст (чего нельзя сделать в $\lambda_\to$).

Квантор ∀ по типам

- Неявно в полиморфной типизации по Карри присутствуют *кванторы всеобщности* по переменным r_j:

$$f : (p \to p) \vdash \lambda g.\lambda z.f(gz) : \forall r.((r \to p) \to r \to p).$$

- По смыслу, $\mathbb{Y} : \forall r.((r \to r) \to r)$, и мы хотим поместить эту декларацию в контекст (чего нельзя сделать в $\lambda_\to$).
- Употребление в контексте типов с кванторами ∀ *на внешнем уровне* разрешается в *системе типов Хиндли – Милнера.*

Квантор ∀ по типам

- Неявно в полиморфной типизации по Карри присутствуют *кванторы всеобщности* по переменным r_j:

$$f : (p \to p) \vdash \lambda g.\lambda z.f(gz) : \forall r.((r \to p) \to r \to p).$$

- По смыслу, $\mathbb{Y} : \forall r.((r \to r) \to r)$, и мы хотим поместить эту декларацию в контекст (чего нельзя сделать в $\lambda_{\to}$).
- Употребление в контексте типов с кванторами ∀ *на внешнем уровне* разрешается в *системе типов Хиндли – Милнера.*
- Однако мы сначала познакомимся с *системой F,* или $\lambda 2$ (типизованное λ-исчисление второго порядка), где квантор разрешается использовать вообще без ограничений.