

# 1 Лекция 9 (30.10.2018)

## 1.1

**Def.** Ранг типа  $m$

$R(x)$  — все типа ранга  $x$ .

- $R(0)$  — все типы без кванторов
- $R(x+1) = R(x) \mid R(x) \rightarrow R(x+1) \mid \forall \alpha. R(x+1)$

**Enddef.**

Например:

- $\alpha \in R(0)$
- $\forall \alpha. \alpha \in R(1)$
- $(\forall \alpha. \alpha) \rightarrow (\forall b. b) \in R(2)$
- $((\forall \alpha. \alpha) \rightarrow (\forall b. b)) \rightarrow b \in R(3)$

Тут видно, если выражение слева от знака импликации имеет ранг  $n$ , то все выражение будет иметь ранг  $\geq (n+1)$ .

**Утверждение:** Пусть  $x$  — выражение только с поверхностными кванторами, тогда  $x \in R(1)$ .

**Def.** Типовая система

$\sigma ::= \forall \alpha_1. \forall \alpha_2. \dots \forall \alpha_n. \tau$ , где  $\tau \in R(0)$  и, следовательно,  $\sigma \in R(1)$ .

**Enddef.**

**Def.** Частный случай (спциализация) типовой схемы

$\sigma_1 \sqsubseteq \sigma_2$  — типовая схема

$\sigma_2$  — частный случай (специализация)  $\sigma_1$ , если

1.  $\sigma_1 = \forall \alpha_1. \forall \alpha_2. \dots \forall \alpha_n. \tau_1$
2.  $\sigma_2 = \forall \beta_1. \forall \beta_2. \dots \forall \beta_m. \tau_1[\alpha_i := S(\alpha_i)]$
3.  $\forall i. \beta_i \in FV(\tau_1)$

**Enddef.**

$M_1 : \forall \alpha. \alpha \rightarrow \alpha$

$M : \forall \beta_1. \forall \beta_2 : (\beta_1 \rightarrow \beta_2) \rightarrow (\beta_1 \rightarrow \beta_2)$

Вполне возможно, что в ходе замены, все типы будут уточнены ( $\alpha$  уточниться как  $\beta_1 \rightarrow \beta_2$ ).

## 1.2 Хиндли-Милнер

1. Все типы только с поверхностными кванторами ( $R(1)$ )
  2.  $\overline{HM} ::= p \mid \overline{HM} \overline{HM} \mid \lambda p. \overline{HM} \mid let = \overline{HM} in \overline{HM}$
- $\exists p. \phi = \forall b. (\forall p. (\phi \rightarrow b)) \rightarrow b$

- $\phi \rightarrow \perp \equiv \forall b.(\phi \rightarrow b)$
- $$\frac{\Gamma, \forall p.(\phi \rightarrow b) \vdash \forall p.(\phi \rightarrow b)}{\Gamma, \forall p.(\phi \rightarrow b) \vdash \phi[p := \Theta] \rightarrow b}$$
- $$\frac{\Gamma, \forall p.(\phi \rightarrow b) \vdash b}{\Gamma \vdash (\forall p.(\phi \rightarrow b)) \rightarrow b}$$
- $$\frac{\Gamma \vdash (\forall p.(\phi \rightarrow b)) \rightarrow b}{\Gamma \vdash \forall b.(\forall p.(\phi \rightarrow b)) \rightarrow b}$$

Соглашение:

- $\sigma$  — типовая схема
- $\tau$  — простой тип

1.  $\overline{\Gamma, x : \sigma \vdash x : \sigma}$
2. 
$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 e_1 : \tau'}$$
3. 
$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$
4. 
$$\frac{\Gamma \vdash e_0 : \sigma \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : \tau}, \text{ let } x = a \text{ in } b \equiv (\lambda x.b) a$$
5. 
$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma}$$
6. 
$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha. \sigma} \alpha \notin FV(\Gamma)$$

### 1.3 Алгоритм вывода типов в системе Хиндли-Милнера W

На вход подаются  $\Gamma, M$ , на выходе наиболее общая пара  $(S, \tau)$

1.  $M = x, x : \tau \in \Gamma$  (иначе ошибка)
  - Выбросить все кванторы из  $\tau$
  - Переименовать все свободные переменные в свежие  
Например:  $\forall \alpha_1. \phi \Rightarrow \phi[\alpha_1 := \beta_1]$ , где  $\beta_1$  — свежая переменная

$(\emptyset, \Gamma(x))$
2.  $M = \lambda n.e$ 
  - $\tau$  — новая типовая переменная
  - $\Gamma' = \Gamma \setminus \{n : \_ \}$  (т.е.  $\Gamma$  без переменной  $n$ )
  - $\Gamma'' = \Gamma' \cup n : \tau$
  - $(S', \tau') = W(\Gamma'', e)$

$(S', S'(\tau) \rightarrow \tau')$

3.  $M = P Q$

- $(S_1, \tau_1) = W(\Gamma, P)$
- $(S_2, \tau_2) = W(S_1(\Gamma), Q)$
- $S_3$  — Унификация  $(S_2(\tau_1), \tau_2 \rightarrow \tau)$

$(S_3 \circ S_2 \circ S_1, S_3(\tau))$

4.  $\text{let } x = P \text{ in } Q$

- $(S, \tau) = W(\Gamma, P)$
- $\Gamma' = \Gamma$  без  $x$
- $\Gamma'' = \Gamma' \cup \{x : \forall \alpha_1 \dots \alpha_k. \tau_1\}$ , где  $\alpha_1 \dots \alpha_k$  все свободные переменные в  $\tau_1$
- $(S_2, \tau_2) = W(S_1(\Gamma''), Q)$

$(S_1 \circ S_2), \tau_2)$

Надеемся, что логика второго порядка противоречива.

## 1.4 Рекурсивные типы

Ранее мы уже рассматривали  $Y$ -комбинатор, но не могли типизировать его и отказывались. Однако в программировании хотелось бы использовать рекурсию, поэтому тут мы введем его аксиоматически.

$Yf =_{\beta} f(Y f)$

$Y : \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha$  — аксиома

И теперь, когда мы хотим написать какую-то рекурсивную функцию, скажем, на языке Ocaml, то интерпретировать ее можно будет следующим образом:

```
let rec f = expr in          let f = Y (\ f. expr) in
  expression                  expression
```

Рекурсивными могут быть не только функции, но и типы. Как, например, список из целых чисел:

```
type intList = Nil | Cons of int * intList;;
```

На нем мы можем вызывать рекурсивные функции, например, ниже представлен фрагмент кода, позволяющий найти длину списка.

```
let rec length l = match l with
| Nil -> 0
| Cons (x, s) -> 1 + length s;;

let my_list = Cons(1, Cons (2, Cons (3, Nil)));;

print_int (length my_list);; (* output: 3 *)
```

Рассмотрим, что из себя представляет тип списка выше:

$Nil = inLeft\ O = \lambda a.\lambda b.a\ O$

$Cons = inRight\ p = \lambda a.\lambda b.b\ p$

$\lambda a.\lambda b.a\ O : \forall \gamma.(\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow \gamma$

$\lambda a.\lambda b.b\ O : \forall \gamma.(\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow \gamma$

$\delta = \forall \gamma.(\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow \gamma$

$\lambda a.\lambda b.b\ (\lambda a.\lambda b.a\ O) : \forall \alpha.(\alpha \rightarrow \gamma) \rightarrow (\delta \rightarrow \gamma) \rightarrow \gamma$

Научимся задавать рекурсивные типы, а именно рассмотрим два способа решения:

## 1. Эквирекурсивный

```
list = Nil | Cons a * list
```

$\alpha = f(\alpha)$  — уравнение с неподвижной точкой. Пусть  $\mu\alpha.f(\alpha) = f(\mu\alpha.f(\alpha))$ . Используем это в типах, а именно  $f$  — это и тип список. То есть мы по сути использовали  $Y$  комбинатор, который для выражений, а для типов ввели аналогичный  $\nu$ .

На практике такой подход используется и в языке программирования Java:

```
class Enum <extends Enum<E>>
```

## 2. Изорекурсивный

В отличие от эквирекурсивных типов будем считать, что  $\mu\alpha.f(\alpha)$  изоморфно  $f(\mu\alpha.f(\alpha))$ . Такой подход используется в языке программирования C.

```
struct list {
    list* x;
    int a;
}
(*x).(*x).(*x).a
// или, что эквивалентно
x->x->x.a
```

Можно заметить, что выше для работы со списком мы использовали специальную операцию:  $* : list* \rightarrow list$  — разыменовывание

В изорекурсивных типах введены специальные операции для работы с этими типами и оператор  $*$  из C это как раз был примером одной из них (в частности roll):

- $Roll : Nil|Cons(a * list) \rightarrow list$
- $Unroll : list \rightarrow Nil|Cons(a * list)$

В более общем виде (введение в типовую систему):

- $roll : f(\alpha) \rightarrow \alpha$
- $unroll : \alpha \rightarrow f(\alpha)$

Можно привести еще пример из языка C:

- $* : T* \rightarrow T$

- $\& : T \rightarrow T^*$
- $T = \alpha$
- $T^* = f(\alpha)$

## 1.5 Зависимые типы

Рассмотрим функцию `sprintf` из языка C:

```
sprintf : string → smth → string
sprintf "%d" : int → string
sprintf "%f" : float → string
```

Легко видеть, что тип `sprintf` определяется первым аргументом. То есть тип этой функции зависит от терма - именно такой тип и называется зависимым (*англ. dependent type*).

Рассмотрим несколько иной пример, а именно список. Предположим, что мы хотим скалярно перемножить два списка:

```
let rec dot lst1 lst2 = match (lst1, lst2) with
| ([], []) -> 0
| (x :: xs, y :: ys) -> x * y + (prod xs ys)
;;
```

```
dot [1; 2] [3; 4] (* results in 11 *)
```

```
dot [1; 2] [3; 4; 5] (* получим ошибку *)
```

Было бы очень здорово уметь отлавливать эту ошибку не в рантайме, а во время компиляции программы и зависимые типы могут в этом помочь. Например в языке Idris можно использовать `Vect`:

```
dot : {n : Nat} -> Vect n Integer -> Vect n Integer -> Integer
dot {n = Z} [] [] = 0
dot {n = (S len)} (x :: xs) (y :: ys) = y * x + dot len xs ys
```

```
let v1 = Data.Vect.fromList [1, 2, 3]
let v2 = Data.Vect.fromList [4, 5, 6]
dot v1 v2 -- results in 32
```

```
let v1 = Data.Vect.fromList [1, 2, 3, 4]
dot v1 v2 -- Type mismatch between
--       Vect 3 Integer (Type of v2)
-- and
--       Vect 4 Integer (Expected type)
```

Если подойти к типу функции `dot` ближе с точки зрения теории типов, то мы бы записали это так (о \* речь пойдет в следующей главе [стоит ее воспринимать как тип типа]):

$Nat : *, Integer : *, Vect : Nat \rightarrow Integer \rightarrow * \vdash dot : \Pi n : Nat. (Vect\ n\ Integer) \rightarrow (Vect\ n\ Integer) \rightarrow Integer$

### 1.5.1 П-типы и $\Sigma$ -типы

- $\Pi x : \alpha. P(x)$  - это запись можно читать как (в каком-то смысле в интуиционистском понимании): "У меня есть метод для конструирования объекта типа  $P(x)$ , использующий любой предоставленный  $x$  типа  $\alpha$ ". Если же смотреть на эту запись с точки зрения классической логики, то ее можно понимать как бесконечную конъюнкцию  $P(x_1) \& P(x_2) \& \dots$ . Данная конъюнкция соответствует декартовому произведению, отсюда и название П-типа (иногда в англоязычной литературе можно встретить *dependent function type*).
- $\Sigma x : \alpha. P(x)$ . Аналогично предыдущему пункту рассмотрим значение с интуиционистской точки зрения: "У меня есть объект  $x$  тип  $\alpha$ , но больше ничего про него не знаю кроме того, что он обладает свойством  $P(x)$ ". Это как раз в стиле интуиционизма, что нам приходится значить и объект  $x$  и его свойство  $P(x)$ . Это можно представить как пару, а пара - бинарное произведение. С точки зрения классической логики, мы можем принимать эту формулу как бесконечную дизъюнкцию  $P(x_1) \vee P(x_2) \vee \dots$ , которая соответствует алгебраическим типам данных. (иногда в англоязычной литературе можно встретить *dependent sum*).

## 2 Лекция 10 (06.11.2019)

### 2.1 Введение

Прежде мы разбирали простотипизированное лямбда исчисление, в котором термы зависели от термов, например, терм  $F M$  зависит от терма  $M$ . После того, как было замечено, что, скажем,  $I$  может иметь разные типы, которые по сути различаются лишь аннотацией, например,  $\lambda x. x : \alpha \rightarrow \alpha$ ,  $\lambda x. x : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ , была введена типовая абстракция, то есть термы теперь могли зависеть от типов и такая типовая система была названа System F и можно было писать  $\lambda \alpha. \lambda x : \alpha. x : \forall \alpha. \alpha \rightarrow \alpha$ . То есть это было своего рода изобретением шаблонов в языке C++. Но на этом все не ограничено. System  $F_w$ , в которой типы могут зависеть от типов, как, например, скажем, список - алгебраический тип данных, у которого есть две альтернативы  $Nil : \forall \alpha. List \alpha$  и  $Cons : \forall \alpha. \alpha \rightarrow List \alpha \rightarrow \alpha$  (рекурсивные типы смотри выше). Для лучшего понимания различия системы  $F$  и  $F_w$  ниже представлены грамматики для типов:

- $T_{\rightarrow} ::= \alpha \mid (T_{\rightarrow}) \mid T_{\rightarrow} \rightarrow T_{\rightarrow}$
- $T_F ::= \alpha \mid \forall \alpha. T_F \mid (T_F) \mid T_F \rightarrow T_F$
- $T_{F_w} ::= \alpha \mid \lambda \alpha. T_{F_w} \mid (T_{F_w}) \mid T_{F_w} \rightarrow T_{F_w} \mid T_{F_w} T_{F_w}$

Ничего не мешает рассматривать типовую систему, в которой тип может зависеть от терма, как это было сделано раньше. Пусть для всех  $a : \alpha$  мы можем определить тип  $\beta_a$  и пусть существует  $b_a : \beta_a$ . Тогда вполне обоснована запись функции  $\lambda a : \alpha. b_a$ . Тип данного выражения приятно записывать как  $\Pi a : \alpha. \beta_a$  (стоит сделать замечание, что если  $\beta_a$  не зависит от  $a$  [то есть функция константа], то вместо  $\Pi a : \alpha. \beta_a$  пишут  $\alpha \rightarrow \beta$ ). Примером может быть тип вектора, длина которого зависит от натурального числа и типа (пример из языка Idris):

```
data Vect : (len : Nat) -> (elem : Type) -> Type where
  Nil  : Vect 0 elem
  (::) : (x : elem) -> (xs : Vect len elem) -> Vect (S len) elem
```

Теперь наша грамматика стало обширной и появилась необходимость более формально говорить о типах, т.е. ввести их в систему. Для этого были придуман род (*англ.* *kind*), который обозначают  $*$ .

Рассмотрим пару примеров, как используется род:

- $\lambda m : \alpha. F \ m : (\alpha \rightarrow \beta) : *$
- $\lambda \alpha : *. I_\alpha : (\Pi \alpha : * : \alpha \rightarrow \alpha) : *$
- $\lambda n : Nat. A^n \rightarrow B : Nat \rightarrow * : *$
- $\lambda \alpha : *. \alpha \rightarrow \alpha : * \rightarrow *$

Рассмотри подробнее последнее выражение, а именно  $* \rightarrow *$ . Это не тип, так как иначе бы могли записать  $* \rightarrow * : *$ , однако понятно, что это не так. В частности для этого вводится понятие сорта (*англ.* *sort*), которое можно воспринимать как тип рода и тогда  $* \rightarrow * : \square$  и  $* : \square$ .

Обобщая все вышесказанное, построим обобщенную типовую систему.

## 2.2 Обобщенная типовая система

- Сорта:  $\{*, \square\}$ 
  - Выражение " $A : *$ " означает, что  $A$  — тип. И тогда, если на метаязыке мы хотим сказать "Если  $A$  тип, то и  $A \rightarrow A$  тоже тип то формально это выглядит как  $A : * \vdash (A \rightarrow A) : *$
  - $\square$  - это абстракция над сортом для типов.
  - Например:

$$\begin{aligned}
 * \ 5 : int : * : \square \\
 * \ [] : * \rightarrow * : \square \\
 * \ \lambda M. List < M > : * \rightarrow * : \square
 \end{aligned}$$

- $T ::= x \mid c \mid T \ T \mid \lambda x : T. T \mid \Pi x : T. T$

- Аксиома:

$$- \overline{\vdash *}. \square$$

- Правила вывода:

1.  $\frac{\Gamma \vdash A : S}{\Gamma, x : A \vdash x : A} \ x \notin \Gamma$
2.  $\frac{\Gamma \vdash A : B \quad \Gamma C : S}{\Gamma, x : C \vdash A : B}$  — правило ослабление (примерно как  $\alpha \rightarrow \beta \rightarrow \alpha$  в И.В.)
3.  $\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : S \quad B =_\beta B'}{\Gamma \vdash A : B'}$  — правило конверсии
4.  $\frac{\Gamma \vdash F : (\Pi x : A. B) \quad \Gamma \vdash a : A}{\Gamma \vdash (F \ a) : B[x := a]}$  — правило применения

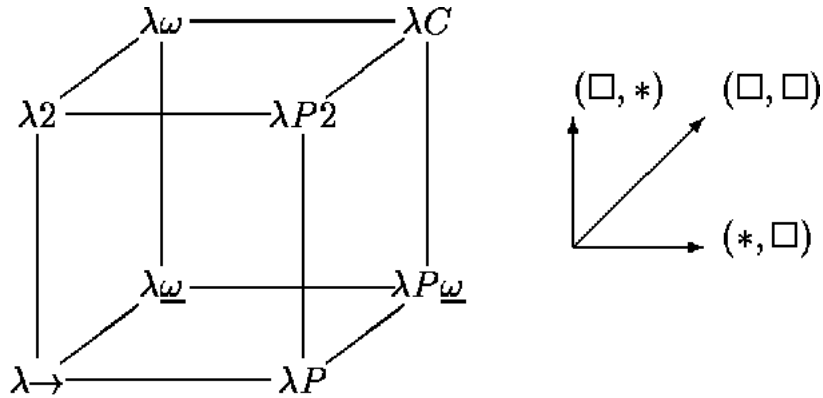
- Семейства правила (generic-правила)

Пусть  $(s_1, s_2) \in S \subseteq \{*, \square\}^2$ .

1. П-правило: 
$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A. B) : s_2}$$
2.  $\lambda$ -правило: 
$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash b : B \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\lambda x : A. b) : (\Pi x : A. B)}$$

## 2.3 $\lambda$ -куб

В обобщенных типовых системах есть generic-правила, которые зависят от выбора  $s_1$  и  $s_2$  из множества сортов. Этот выбор можно проиллюстрировать в виде куба.



Выбор правил означает следующее:

- $(*, *)$  - позволяет записывать термы, которые зависят от термов
- $(\square, *)$  - позволяет записывать термы, которые зависят от типов
- $(*, \square)$  - позволяет записывать типы, которые зависят от термов
- $(\square, \square)$  - позволяет записывать типы, которые зависят от типов

Также на этом кубике можно расположить языки программирования, например:

- Haskell будет располагаться на левой грани куба, недалеко от  $\lambda\omega$
- Idris и Coq, очевидно, будет находиться в  $\lambda C$
- C++ очень ограниченно приближается к  $\lambda C$  (мысли вслух):
  1.  $(*, *)$  - без этого не может обойтись ни один язык программирования
  2.  $(\square, *)$  - например, `sizeof(type)`
  3.  $(*, \square)$  - например, `std::array<int, 19>` - тут есть ограничение на то, значение каких типов можно подставлять.
  4.  $(\square, \square)$  - например, `std::vector<int>`, `int*`



## 2.4 Свойства

Для систем в  $\lambda$ -кубе верны следующие утверждения:

- **Th. SN** Обобщенная типовая система сильно нормализуема
  1. Для любых двух элементов  $A, B$  и  $C$ , таких,  $A \twoheadrightarrow B$  и  $A \twoheadrightarrow C$  верно, что существует  $D$ , что  $B \twoheadrightarrow D$  и  $C \twoheadrightarrow D$
  2. Для любых двух элементов  $A, B$ , для которых верно  $A =_\beta B$ , существует  $C$ , что  $A \twoheadrightarrow C$  и  $B \twoheadrightarrow C$
- **Th. Черча-Россера**
- **Th. Subject reduction**  $\Gamma \vdash A : T$  и  $A \twoheadrightarrow B$ , тогда  $\Gamma \vdash B : T$
- **Th. Unicity of types**  $\Gamma \vdash A : T$  и  $\Gamma \vdash A : T'$  тогда  $T =_\beta T'$

Примеры:

- $\lambda\omega$ :

$$\vdash (\lambda\alpha : *. \alpha \rightarrow \alpha)(* \rightarrow *) : \square$$

$$\begin{array}{c}
 1. \frac{\vdash * : \square \quad \frac{\vdash *. \square}{a : * \vdash *. \square}}{\vdash (* \rightarrow *) : \square} \\
 \\
 2. \frac{\vdash * : \square \quad \frac{\alpha : * \vdash \alpha : * \quad \alpha : *, x : * \vdash \alpha : *}{\alpha : * \vdash \alpha \rightarrow \alpha : x} \quad \frac{\vdash * : \square \quad \frac{\vdash *. \square}{a : * \vdash *. \square}}{\vdash (* \rightarrow *) : \square}}{\vdash (\lambda\alpha : *. \alpha \rightarrow \alpha) : * \rightarrow *}
 \end{array}$$

Notes:

- $(\lambda x.x) : (A \rightarrow A)$  - implicit typing (Curry style)
- $I_A = \lambda x : A. x$  - explicit typing (Church style)