1 Лекция 9 (30.10.2018)

1.1

Def. Ранг типа m

$R(x)$ — все типа ранга x.

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

Enddef.

Например:

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

Тут видно, если если выражение слева от знака имликации 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

σ_2 — частный случай (специализация) σ_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.

$$M_1 : \forall\alpha.\alpha \to \alpha$$

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

Вполне возможно, что в ходе замены, все типы будут уточнены (α уточниться как $\beta_1 \to \beta_2$.

1.2 Хиндли-Милнер

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

- $\exists p.\phi = \forall b.(\forall p.(\phi \to b)) \to b$

- $\phi \to \bot \equiv \forall b.(\phi \to b)$

- $$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma, \forall p.(\phi \to b) \vdash \forall p.(\phi \to b)}{\Gamma, \forall p.(\phi \to b) \vdash \phi[p := \Theta] \to b}}{\Gamma, \forall p.(\phi \to b) \vdash b}}{\Gamma \vdash (\forall p.(\phi \to b)) \to b}}{\Gamma \vdash \forall b.(\forall p.(\phi \to b)) \to b}$$

Соглашение:

- σ — типовая схема

- τ — простой тип

1. $\dfrac{}{\Gamma, x : \sigma \vdash x : \sigma}$

2. $\dfrac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'}$

3. $\dfrac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$

4. $\dfrac{\Gamma \vdash e_0 : \sigma \qquad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash let\ x = e_0\ in\ e_1 : \tau}$, $let\ x = a\ in\ b \equiv (\lambda x.b)\ a$

5. $\dfrac{\Gamma \vdash e : \sigma' \qquad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma}$

6. $\dfrac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha.\sigma}$ $\alpha \notin FV(\Gamma)$

1.3 Алгоритм вывода типов в системе Хиндли-Милнера W

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

1. $M = x,\ x : \tau \in \Gamma$ (иначе ошибка)

   - Выбросить все кванторы из τ
   - Переименовать все свободные переменные в свежие
     Например: $\forall \alpha_1.\phi \Rightarrow \phi[\alpha_1 := \beta_1]$, где β_1 — свежая переменная

   $(\emptyset, \Gamma(x))$

2. $M = \lambda n.e$

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

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

3. $M = P\ Q$

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

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

4. $let\ x = P\ in\ Q$

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

$Y f =_\beta f(Y\ f)$

$Y : \forall \alpha . (\alpha \to \alpha) \to \alpha$ — 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
    expression
```

```
let f = Y (λ f. expr) in
    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:
$Nil = inLeft\ O = \lambda a.\lambda b.a\ O$
$Cons = inRight\ p = \lambda a.\lambda b.b\ p$
$\lambda a.\lambda b.a\ O : \forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$
$\lambda a.\lambda b.b\ O : \forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$
$\delta = \forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$
$\lambda a.\lambda b.b\ (\lambda a.\lambda b.a\ O) : \forall\alpha.(\alpha \to \gamma) \to (\delta \to \gamma) \to \gamma$

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 ν.

   На практике такой подход используеься и в языке программирования Java:

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

   Также приведем пример вывода типа $\lambda x.x\ x$ (можно вспомнить, что именно этот терм помешал нам типизировать Y-комбинатор в простотипизированном λ-исчислении):

   Пусть $\tau = \mu\alpha.\alpha \to \beta$. Если мы раскроем τ один раз, то получим $\tau = \tau \to \beta$. Если раскроем еще раз, то получим $\tau = (\tau \to \beta) \to \beta$.

$$\dfrac{x:\tau \vdash x:\tau\to\beta \qquad x:\tau \vdash x:\tau}{\dfrac{x:\tau \vdash x\ x:\beta}{\vdash \lambda x.x\ x:\tau\to\beta}}$$

   Ранее мы ввели Y-комбинатор аксиоматически, а можем ли мы его типизировать используя рекурсивные типы? Ответ: Да, можем. Напомним, что $Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\lambda f:\beta\to\beta,\ x:\tau\vdash f:\beta\to\beta \qquad f:\beta\to\beta,\ x:\tau\vdash x\ x:\beta}{f:\beta\to\beta, x:\tau\vdash f\ (x\ x)}}{f:\beta\to\beta\vdash\lambda x.f\ (x\ x):\tau}}{\lambda f:\beta\to\beta\vdash\lambda x.f\ (x\ x):\tau\to\beta} \qquad \dfrac{\text{аналогично в другой ветке}}{\lambda f:\beta\to\beta\vdash\lambda x.f\ (x\ x):\tau}}{\dfrac{f:\beta\to\beta\vdash(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)):\beta}{\dfrac{\vdash\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)):(\beta\to\beta)\to\beta}{\vdash\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)):\forall\beta.(\beta\to\beta)\to\beta}}}$$

   Загадочка: А можно ли типизировать, скажем $\lambda x : Nat.x(Sx)$?

2. Изорекурсивный

   В отличии 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: $* : list* \to list$ — разыменовывание

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

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

В более общем виде (введение в типовую систему):

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

Можно привести еще пример из языка C:

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

1.5 Зависимые типы

Рассмотрим функцию sprintf из языка C:

$sprintf : string \to smth \to string$

$sprintf"\%d" : int \to string$

$sprintf"\%f" : float \to string$

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 + (dot 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 завсимые 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 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]):

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

1.5.1 Π-типы и Σ-типы

- $\Pi x : \alpha.P(x)$ - это запись можно читать как (в каком-то смысле в интуционистском понимании): "У меня есть метод для конструирования объекта типа $P(x)$, использующий любой предоставленный x типа α". Если же смотреть на эту запись с точки зрения классической логики, то ее можно понимать как бесконечную конъюнкцию $P(x_1)\&P(x_2)\&...$. Данная конъюнкция соответствует декартовопу произведению, отсюда и называние П-типа (иногда в англоязычной литературе можно встретить *dependent function type*).

- $\Sigma x : \alpha.P(x)$. Аналогично предыдущему пункту рассмотрим значение с интуционистской точки зрения: "У меня есть объект x тип α, но больше ничего про него не знаю кроме того, что он обладает свойством $P(x)$". Это как раз в стиле интуционизма, что нам приходится значть и объект x и его свойство $P(x)$. Это можно представить как пару, а пара - бинарной произведение. С точки же зрения классической логики, мы можем принимать эту формулу как бесконечную дизъюнкицю $P(x_1) \vee P(x_2) \vee ...$, 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 зависили 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, $\lambda x.x : \alpha \to \alpha$, $\lambda x.x : (\alpha \to \alpha) \to (\alpha \to \alpha)$, была введена типовая абстракция, то есть термы теперь могли зависеть от типов и такая типовая система была названа System F и можно было писать $\Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha \to \alpha$. То есть это было своего рода изобретением шаблонов в языке C++. Но на этом все не ограничено. System F_w, в которой типы могут зависить от типов, как, например, скажем, список - алгебраический тип данных, у которого есть две альтернативы $Nil : \forall \alpha. List \alpha$ и $Cons : \forall \alpha. \alpha \to List \alpha \to \alpha$ (рекурсивные типы смотри выше). Для лучшего понимания различия системы F и F_w ниже представлены грамматики для типов:

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

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 β_α и пусть существует $b_\alpha : \beta_\alpha$. Тогда вполне обоснована запись функции $\lambda \alpha : b_\alpha$. Тип данного выражения приянто записывать как $\Pi a : \alpha. \beta_\alpha$ (стоит сделать замечание, что если β_α не зависит от α [то есть функция константа], то вместо $\Pi a : \alpha. \beta_\alpha$ пишут $\alpha \to \beta$). 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 Z 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:

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

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

Обобщая все вышесказанное, построим обобщенную типовую систему.

2.2 Обобщенная типовая система

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

– Например:

  * $5 : int : * : \Box$
  * $[] : * \to * : \Box$
  * $\Lambda M.List < M >: * \to * : \Box$

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

- Аксиома:

  – $\overline{\vdash *.\Box}$

- Правила вывода:

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

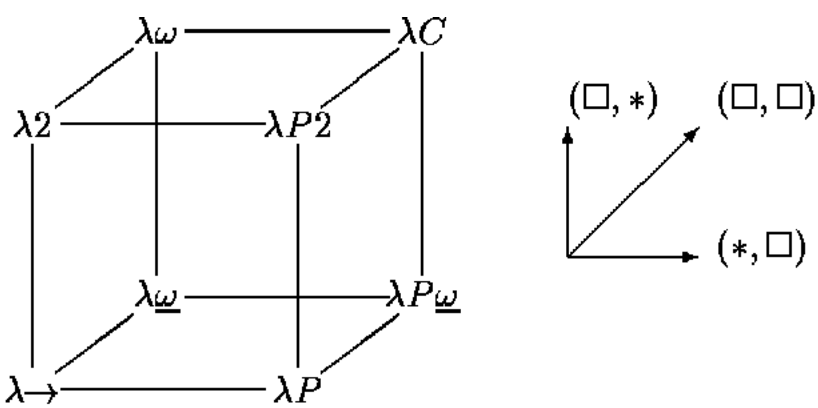
- Семейства правила (generic-правила)

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

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

2.3 λ-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 распологаться на левой грани куба, недалеко от λw
- Idris и Coq, очевидно, будет находиться в λC
- C++ очень ограниченно приближается к λC (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 λ-кубе верны следующие утверждения:

- **Th. SN** Обобщенная типовая система сильно нормализуема
- **Th. Черча-Россера**
  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. 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:

- $\lambda\omega$:

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

$$1.\ \frac{\vdash * : \square \qquad \dfrac{\vdash *.\square}{a : * \vdash *.\square}}{\vdash (* \to *) : \square}$$

$$2.\ \frac{\vdash * : \square \qquad \dfrac{\alpha : * \vdash \alpha : * \qquad \alpha : *, x : \alpha \vdash \alpha : *}{\alpha : * \vdash \alpha \to \alpha : x} \qquad \dfrac{\vdash * : \square}{a : * \vdash * : \square}}{\vdash (\lambda\alpha : *.\alpha \to \alpha) : * \to *}$$

Notes:

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