# Теория типов, интуиционистская логика и колмогоровская сложность.

Даня Рогозин

ИФ РАН

Май, 2017

# Введение.

- Теория сложности доказательств занимается (в том числе) оценками алгоритмической сложности длины доказательства формулы в том или ином исчислении.
- Идея: рассмотреть понятие сложности доказательства в несколько ином ракурсе, предположим у нас есть формула $\phi$, у которого есть доказательство $A$. Процесс доказательства является некоторой процедурой, состоящей из последовательности действий, заданных правилами (алгоритмом). Мы полагаем, что $\phi$ является кодом данной программы, доказывающей данную формулу. Ясно также, что данный код написан в некотором языке, который порожден нужной грамматикой, предназначенная для описания наших программ-доказательств.
- Вопрос: можно ли рассматривать сложность доказательства формулы через сложность описания алгоритма, который ее доказывает?
- Ответ: можно попробовать.

# Колмогоровская сложность.

- Декомпрессор (способ описания) — это частично определенная вычислимая функция одного аргумента, $D : \Xi \to \Xi$, сопоставляющая некоторым способом битовой строке другую битовую строку. Пусть $x, y \in \Xi$, тогда обозначение $D(x) = y$ говорит, что $x$ является описанием $y$ при заданном декомпрессоре $D$
- Колмогоровской сложностью битовой строки $x$ при декомпрессоре $D$ будет минимальной длиной такой строки $y$, что $D(y) = x$:

$$KS_D(x) = min\{l(y) \mid D(y) = x\} \tag{1}$$

  где $l(y)$ — длина строки $y$.
- Существует оптимальный декомпрессор $D$, такой что, для для любого другого декомпрессора $D'$ найдется $c$, что $KS_D(x) \leq KS_{D'}(x) + c$.
- $KS$ невычислима, но перечислима сверху.

# Условная сложность и сложность пары.

- Условной сложностью ($KS_D(z|y)$) слова $z$ при условии $y$ при фиксированном способе условного описания $D$ мы называем длину кратчайшего описания $z$ при условии $y$:

$$KS_D(z|y) = min\{l(x) \mid D(x, y) = z\}. \quad (2)$$

- $KS(x, y) = KS(x) + KS(y|x) + O(\log N)$.
- Оптимальность распространяется и на условную сложность.

# Двоичное лямбда-исчисление.

- Разработано и реализовано Джоном Тромпом в начале нулевых;
- Реализовано за счет двоичного кодирования термов де Брюйна:
  1. $\widehat{n} := 1^{n+1}0$;
  2. $\widehat{\lambda M} := 00\widehat{M}$;
  3. $\widehat{(M\ N)} := 01\widehat{M}\widehat{N}$.
- Пример:

$$\texttt{SUCC} := \lambda n.\lambda f.\lambda x.f(nfx) \Rightarrow \lambda\lambda\lambda 1(210) \Longrightarrow 000000011100111100111010 \qquad (3)$$

# Двоичное лямбда-исчисление.

- Имея данное представление лямбда-исчисления мы можем задаться вопросом о сложности размера программы;
- Ограничимся верхней оценкой для произвольного лямбда-терма:
  $KS(\widehat{M}) \leq |x| + |\widehat{\mathbf{I}}| = |x| + |\widehat{\lambda x.x}| = |x| + |0010| = |x| + 4.$
- Вообще оценки можно варьировать, поскольку в данном лямбда-исчислении есть самоинтерпретатор, реализованный засчет существования комбинаторов неподвижной точки. Если хотим типы, то нужно думать.

# Двоичное лямбда-исчисление.

- Имея данное представление лямбда-исчисления мы можем задаться вопросом о сложности размера программы;
- Ограничимся верхней оценкой для произвольного лямбда-терма: $KS(\widehat{M}) \leq |x| + |\widehat{\mathbf{I}}| = |x| + |\widehat{\lambda x.x}| = |x| + |0010| = |x| + 4$.
- Вообще оценки можно варьировать, поскольку в данном лямбда-исчислении есть самоинтерпретатор, реализованный засчет существования комбинаторов неподвижной точки. Если хотим типы, то нужно думать.
- Первая задача — реализовать алгоритм, который по типизируемуму терму вернет нам тип и двоичный код:

# Двоичное лямбда-исчисление и типы.

Реализация бестипового лямбда-исчисления и термов де Брюйна:

```
type term = Var of string | Abs of string * term | App of term * term

type debruijn = Variable of int | Abstraction of debruijn
              | Application of debruijn * debruijn

val termToString : term -> string = <fun>

val debruijnize : term -> debruijn = <fun>
```

# Двоичное лямбда-исчисление и типы.

Двоичное кодирование «дебрюйнизированных» лямбда-термов:

```
let rec numberCode = function
  | 0 -> "10"
  | n -> "1" ^ (numberCode (n - 1))

let rec debruijnToBinary = function
 | Variable(i) -> numberCode i
 | Abstraction(t) -> "00" ^ (debruijnToBinary t)
 | Application(t, u) -> "01" ^ (debruijnToBinary t) ^ (debruijnToBinary u)

let termToBinary term = debruijnToBinary (debruijnize term)
```

# Двоичное лямбда-исчисление и типы.

Теперь мы начнем реализовывать систему типов:

Введем сначала грамматику для термов и для типов:

```
type ty = TyBool | TyNat | TyArrow of ty * ty | TyError

type typedTerm = TmFalse | TmTrue | TmMrBean
               | TmBear | TmZero | TmSucc of typedTerm
               | TmIf of typedTerm * typedTerm * typedTerm
               | TmVar of string * int
               | TmAbs of string * ty * typedTerm
               | TmApp of typedTerm * typedTerm
```

# Двоичное лямбда-исчисление и типы.

Далее, объявления для контекстов и полезные функции для работы с ними:

```
type binding = NameBind | VarBind of ty

type context = (string * binding) list

let addbinding ctx x bind = (x,bind)::ctx

let getbinding ctx n = snd (List.nth ctx n)

let getTypeFromContext ctx n =
  match getbinding ctx n with
    VarBind(tyT) -> tyT
  | _ -> TyError

val typeof : (string * binding) list -> typedTerm -> ty = <fun>
```

# Двоичное лямбда-исчисление и типы.

Первая функция берет типизированный терм и возвращает бестиповый, а вторая выводит строковое сообщение о типизации терма и об ассоциировании с ним нужной двоичной строки.

```
let rec erase = function
    TmVar(var, number) -> Var(var)
  | TmAbs(x, tyT1, t2) -> Abs(x, erase t2)
  | TmApp(t1, t2) -> App(erase t1, erase t2)
  | TmFalse -> fls
  | TmTrue -> tru
  | TmMrBean -> pair
  | TmBear -> returnFirst
  | TmZero -> zero
  | TmSucc n -> App(succ, erase n)
  | TmIf(t1, t2, t3) -> App(App(App(test, erase t1), erase t2), erase t3)

let binaryTermHasType x ctx = termToString (erase x) ^ " :: " ^
        printType (typeof ctx x) ^ " which is " ^ termToBinary (erase x)
```

# Двоичное лямбда-исчисление и типы.

Вывод типовых объявлений в консоль для списка термов `mainList`:

```
let uncurry f (x,y) = f x y

let typeDeclareForBinaries xs = List.map (uncurry binaryTermHasType) xs

let printBinType = List.map print_endline (typeDeclareForBinaries mainList)
```

# Итог, скрин из терминала

```
λx. λy. x :: (a -> (b -> a)) which is 0000110
(f a) :: b which is 011110110
λx. (f x) :: (a -> b) which is 0001111010
λf. λx. (f x) :: ((a -> b) -> (a -> b)) which is 00000111010
λf. λg. λx. ((f x) (g x)) :: ((a -> (b -> c)) -> ((a -> b) -> (a -> c))) which is 0000000101111010011101O
λf. λg. λx. (g (f x)) :: ((a -> b) -> ((b -> c) -> (a -> c))) which is 0000000111001111010
λf. λx. λy. ((f y) x) :: ((a -> (b -> c)) -> (b -> (a -> c))) which is 0000000101111010110
λf. λx. ((f x) x) :: ((a -> (a -> b)) -> (a -> b)) which is 000001011101010
(λx. (x x) λx. (x x)) :: error which is 010001101000011010
λn. λm. λf. λx. ((n f) ((m f) x)) :: ((b -> (b -> b)) -> ((b -> (b -> b)) -> (b -> (b -> b)))) which is 0000000001011111011001011111011010
λn. λm. λf. λx. ((n (m f)) x) :: ((b -> (b -> b)) -> ((b -> b) -> (b -> (b -> b)))) which is 000000000101111100111101101O
val printBinType : unit list = [O; O; O; O; O; O; O; O; O; O; O; O]
#
```

## Попытка верхней оценки сложности типа.

- Тип заселяется термом единственным образом, поэтому мы можем использовать для верхней оценки сложности типа верхнюю оценку для сложности терма:

$$KS(\phi) = KS(\hat{M}) \leq l(\hat{M}) + 4 \tag{4}$$

- Тогда следующие типы будут иметь следующие верхние оценки:
- $KS(\alpha \to \alpha) = KS(\widehat{\mathtt{I}}) \leq KS(\widehat{\lambda x.x}) \leq l(\widehat{\mathtt{I}}) + 4 = l(0010) + 4 \leq 8$;
- $KS((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma) = KS(\widehat{\mathtt{S}}) = KS(\lambda x.\lambda y.\widehat{\lambda z.(xy)}(xz)) \leq l(\widehat{\mathtt{S}}) + 4 = l(00000001011111011001111010) + 4 = 25 + 4 = 29 \leq 29$;
- $KS(\alpha \to \beta \to \alpha) = KS(\widehat{\mathtt{K}}) \leq l(0000110) + 4 \leq 11$;
- $KS(\alpha \to \beta \to \gamma) = KS(\widehat{\mathtt{FALSE}}) \leq l(000010) + 4 \leq 10$;
- Вопрос: есть ли тип, KS которого выползет за данную оценку?
- Данные оценки получаются через прямое вычисление длины кода лямбда-терма, что хорошо, но не позволяет нам анализировать сложность типа с учетом характера задачи, которую он решает.

# Колмогоровская сложность и сложность задач.

- Результаты Верещагина и Шеня (начало нулевых):

# Колмогоровская сложность и сложность задач.

- Результаты Верещагина и Шеня (начало нулевых):
- Пусть $A$, $B$ — некоторые множества двоичных строк (которые мы будем называть задачи), мы введем следующие операции над данными множествами:
  $A \to B = \{p \mid \forall x \in A, [p](x) \in B\}$, где $[p]$ — это (геделев) номер программы $p$, $[p](x)$ — это результат применения программы $p$ к входу $x$;
  $A \wedge B = \{\langle x, y\rangle \mid x \in A, y \in B\}$ (нетрудно видеть, что данное определение совпадает с определением декартова произведения);
  $A \vee B = \{\langle 0, x\rangle \mid x \in A\} \cup \{\langle 1, y\rangle \mid y \in B\}$.
- Не так трудно догадаться, что данный подход основан идеи реализуемости по Клини и на BHK-семантике.

# Колмогоровская сложность и сложность задач.

- Результаты Верещагина и Шеня (начало нулевых):
- Определим сложности задач при зафиксированном оптимальном декомпрессоре $D$, пусть $A = \{x\}$ и $B = \{y\}$ для простоты будут синглтонами:
  $KS(A \wedge B) = KS(x, y) = KS(x) + KS(y|x) + O(\log N)$;
  $KS(A \vee B) = min\{x, y\} + O(1)$;
  $KS(A \to B) = KS(y|x) + O(1)$.
- Примеры:
  $KS(A \to (B \to C)) = KS(C|A, B)$. Поскольку $KS(A \to (B \to C)) = KS((A \wedge B) \to C)$, а $KS((A \wedge B) \to C) = KS(C|A, B)$. Вообще данный пример можно обобщить аналогичным рассуждением:
  $KS(A_1 \to (A_2 \to ... \to (A_{n-1} \to A_n))) = KS(A_n|A_1, A_2, ..., A_n)$;
  $KS((A \to B) \to C) = \min\{KS(C), KS(A) + KS(C|A, B)\}$;
  $KS(A \to A) = KS(A|A) + O(1) = KS(A) + O(1)$;
  $KS((A \to B) \to A) \to A) = 0$.

# Колмогоровская сложность и сложность задач.

Импликация обладает хорошим свойством:

$$KS(B) \leq KS(A \to B) + KS(A) + O(\log N) \tag{5}$$

Пусть $A = \{a\}, B = \{b\}$. По определению сложности пары, $KS(a, b) = KS(a) + KS(b|a) + O(\log N)$. Тогда $KS(a, b) \leq KS(a) + KS(b|a)$. По свойству сложности пары, $KS(b) \leq KS(a, b)$. Тогда $KS(b) \leq KS(a) + KS(b|a) + O(\log N)$, или $KS(B) \leq KS(A) + KS(A \to B) + O(\log N)$.

# Колмогоровская сложность и сложность задач.

Сложность пары (теорема Колмогорова-Левина) позволяет нам получить сложность импликации через сложность условия и решения. Ибо,
$KS(A, B) = KS(A) + KS(A \to B) + O(\log N)$, откуда мы просто выражаем сложность импликации: $KS(A \to B) = KS(A, B) - KS(A) + O(\log N)$.

- Попробуем применить данный подход к соответствию Карри-Говарда (системы $\lambda_{\to}$), рассматривая типы как задачи, решениями которых будут двоичные коды лямбда-термов (что мы и набросали в Ocaml).
- Тогда оценки с точностью до $O(1)$ будут одни и те же (сложность инвариантна по отношению к способу кодирования).
- Иными словами, следуя оптимальности, нам неважно, какой код использовать, чтобы оценить сложность произвольной задачи.
- В случае с $\lambda_{\to}$ есть одно преимущество: мы можем оценивать сложности описания для термов, которые являются кодами для $\lambda_{\to}$-представимых вычислимых функций, используя, в частности, тот факт, что
$KS(A_1 \to (A_2 \to ... \to (A_{n-1} \to A_n))) = KS(A_n | A_1, A_2, ..., A_n)$.

# $\lambda_{\to}$-определимые функции.

Класс $\lambda_{\to}$-определимых функций совпадает с классом расширенных многочленов:
Класс расширенных многочленов — это замнкутный относительно композиции наименьший класс функций над $\mathbb{N}$, которые содержат:

- константные функции 0 и 1;
- проекции вида $U(x_1, ..., x_n)_n^i = x_i$;
- сложение;
- умножение;
- условный оператор $\texttt{ifZeroThen}(x, y, z) = \texttt{if } x = 0 \texttt{ then } y \texttt{ else } z$.

1. Константы 0 и 1 имеют вид $\lambda x.\lambda y.y$ и $\lambda f.\lambda x.fx$. Произвольное натуральное число $n$ имеет вид $\lambda f.\lambda x.f(...(fx)...)$, где $f$ применяется к $x$ $n$ раз.
2. Функция проекции выразима как $\lambda x_1...\lambda x_n.x_i, i \in \{1, ...n\}$.
3. Функция сложения имеет вид $\lambda n.\lambda m.\lambda f.\lambda x.nf(mfx)$.
4. Функция умножения: $\lambda n.\lambda m.\lambda f.\lambda x.n(mf)x$.
5. Функция $\texttt{ifZeroThen}(x, y, z)$ представлена как $\lambda n.\lambda m.\lambda p.\lambda f.\lambda x.n(\lambda y.pfx)(mfx)$

# $\lambda_{\to}$-определимые функции.

Ясно, что данные термы типизируемы, рассмотрим для примера функцию композиции:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f:\beta\to\gamma\vdash f:\beta\to\gamma}\quad\dfrac{\dfrac{}{g:\alpha\to\beta\vdash g:\alpha\to\beta}\quad\dfrac{}{x:\alpha\vdash x:\alpha}}{g:\alpha\to\beta,x:\alpha\vdash gx:\beta}}{f:\beta\to\gamma,g:\alpha\to\beta,x:\alpha\vdash f(gx):\gamma}}{f:\beta\to\gamma,g:\alpha\to\beta\vdash\lambda x.f(gx):\alpha\to\gamma}}{f:\beta\to\gamma\vdash\lambda g.\lambda x.f(gx):(\alpha\to\beta)\to\alpha\to\gamma}}{\vdash\lambda f.\lambda g.\lambda x.f(gx):(\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}$$

# $\lambda_{\to}$-определимые функции.

Пример: сложность композиции.
Для читаемости перепишем наш терм в синтаксисе a-la Haskell:

```
(.) : (b -> c) -> (a -> b) -> a -> c
f . g = \ x -> f (g x)
```

$KS((\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma) = KS(\widehat{f(gx)}|\hat{f}, \hat{g}, \hat{x}) =$
$KS(\widehat{f(gx)}, \hat{f}, \hat{g}, \hat{x}) - KS(\hat{f}, \hat{g}, \hat{x}) + O(\log N)$.
Иными словами, сложность самой задачи с логарифмической точностью равна разности набора «решение + условие» и «набора условие».
Для каждого базовго расширенного многочлена и для композитора можно указанным выше или подобным образом оценить сложность его описания.
TODO: рассмотреть, можно ли дать обобщенную оценку для расширенного многочлена, утоняющую верхнюю оценку для произвольного терма (и типа).

# Спасибо за внимание!