План.

1. Что дают типы на практике и в теории (общие соображения).
2. Мотивация для введения типов в лямбда-исчисления. Простое типизированное лямбда-исчисление.
3. Полиморфное лямбда-исчисление (система F).
4. Полиморфное лямбда-исчисление с зависимыми типами.
5. Реализация обеих систем типов в функциональных языках программирования (Haskell, Idris).
6. Благодарность за внимание.

Что дают типы на практике и в теории.

Общие соображения можно посмотреть, например, в книге Бенджамина Пирса «Типы в языках программирования»:
0. Классификация объектов в соответствии с их свойствами и возможными операциями над ними.
1. Частичная спецификация программы (что содержит некоторую информацию о ее поведении);
2. Проверка типов может отлавливать простые ошибки (во многих языках вбрасывается исключение на подобие `TypeError`, если, скажем, вместо целого числа на вход программы был подан литерал: `unsupported operand type(s) for +: 'int' and 'str'`).
А также:
3. Основа не-теоретико-множественных оснований математики.

Простое типизированное лямбда-исчисление.

0. $Term ::= Var_\lambda \mid (Var_\lambda Var)_\lambda \mid \lambda Var_\lambda . Var_\lambda$
1. $\Phi = \{\alpha, \beta, \gamma, ...\}$; $\phi, \psi \in \Phi \Rightarrow \phi \to \psi$;
2. $\Gamma = \{x : \phi \mid x \in Var_\lambda, \phi \in \Phi\}$;
3. $x : \phi \in \Gamma := \Gamma \vdash x : \phi$;
4. Конструктор и элинимантор для стрелочного типа:

$$\frac{x : \phi \vdash M : \psi}{\Gamma \vdash (\lambda x.M) : (\phi \to \psi)}(In_\to) \quad \frac{\Gamma \vdash f : \phi \to \psi \quad \Gamma \vdash x : \phi}{\Gamma \vdash fx : \psi}(El_\to) \tag{1}$$

5. Примеры объявлений:

5.1. $\lambda f.\lambda g.\lambda x.(fx)(gx) : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$;

5.2. $\lambda f.\lambda g.\lambda x.f(gx) : (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$.

Остальные конструкторы и их элиминаторы задаются в сообразно с правилами введения и удаления соответствующих связок в интуиционистком естественном выводе.

Лямбда-исчисление с кайндами.

Мотивация - введение абстркации и аппликации над типами, то есть ввести над типами систему типов. (НУЖНО БОЛЬШЕ ТИПОВ!!!)

1. Заданим кайнды (сорты типов):

$$K ::= * \mid K \to K \tag{2}$$

2. Правило «кайндизации»:

$$\frac{\Gamma \vdash \phi : (\tau \to *) \quad \Gamma \vdash N : \tau}{\Gamma \vdash \phi N : *} \tag{3}$$

Если базовые типы данных имеют простой кайнд:

$$Nat : *, Bool : *, \ldots \tag{4}$$

то операторы на типами имеют стрелочный кайнд, например тип декартова произведения:

$$\langle , \rangle : * \to * \to * \tag{5}$$

Полиморфное лямбда-исчисление.

Мы будем отталкиваться от системы F, предложенной Жаном-Ивом Жираром в начале 1970-х. Основная модификация - это возможность универсальной квантификации над типами, что позволяет рассматривать полиморфизм (возможность объявить функцию, которую можно вызвать для произвольных типов):

1. Типы в системе F:

$$T ::= V \mid T \to T \mid \forall V.T \tag{6}$$

2. Конструктор и элиминатор для $\forall$ (конструктор и элиминатор для стрелочного типа аналогичны тем, что в простой теории типов):

$$\frac{\Gamma \vdash M : \psi}{\Gamma \vdash M : \forall \alpha.\psi}(Gen) \quad \frac{\Gamma \vdash M : \forall \alpha.\psi}{\Gamma \vdash M : \psi[\alpha := \phi]} \tag{7}$$

3. Смысл $\forall \alpha.\psi$ — каждому типу ϕ мы сопоставляем терм типа $\psi[\alpha := \psi]$.
4. Примеры объявлений:

3.1. $\lambda f.\lambda g.\lambda x.(fx)(gx) : \forall \alpha\beta\gamma : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$;

3.2. $\lambda f.\lambda g.\lambda x.f(gx) : \forall \alpha\beta\gamma : (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$.

Зависимая типизация.

Общая идея — это зависимость значения объявленного типа данных, который зависит от другого типа данных. Данная идея довольно активно используется в практическом программировании. Пример из Java, построение параметризованного класса:

```
public class Main {
  public static void main(String[] args)" {
    ArrayList<Integer> list = new ArrayList<Integer>();
    populateNumbers(list);
    int total = 0;
    for(Integer val : list)
    {
    total = total + val;
    }
  System.out.println(total);
  }
}
```

Мотивация для зависимых типов.

Тип может зависеть от типового параметра, как в примере выше, а может зависеть от значения терма некоторого типа, именно такие типы называются зависимыми. Но как мы увидим далее, мы, по факту имеем дело с разными видами зависимости в разных случаях типизации.

Зависимая типизация (именно в этом узком смысле) также используется неявно, здесь хорошим примером послужить следующий фрагмент кода, написанный на C++:

```
int main(){
  int n;
  cin » n;
  vector int<a>;
  for (i = 0, i < n, i = i - 1){
    cin » a[i];
  ...
  return 0;
 }
}
```

Мотивация для зависимых типов.

Например, списки, неупорядоченные последовательности элементов одного типа, могут иметь сколь угодно большую длину. Для перехода от списка к векторам необходимо, чтобы в типе содержалась информация о его длине, то есть нам мало того, что у данного конструктора есть стрелочный кайнд. Тогда мы ставим наш тип данных в зависимость от терма, положительной целочисленной переменной в данном случае:

$Vector : Nat \to \alpha \to Type\ where$

$Nil : Vector\ 0\ \alpha$

$(::) : \Pi n : \mathbb{N}.\alpha \to Vector\ n\ \alpha \to Vector\ (S\ n)\ \alpha$

Другие примеры объявлений в таком стиле:

$Head : \Pi n : \mathbb{N}.Vector\ (S\ n)\ \alpha \to \alpha$

$Tail : \Pi n : \mathbb{N}.Vector\ (S\ n)\ \alpha \to Vector\ n\alpha$

Зависимая типизация.

Мы для простоты будем отталкиваться от системы $\lambda P\omega$:

1. Конструктор и элиминатор для зависимого произведения:

$$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x.M) : (\Pi x : A.B)} \quad \frac{\Gamma \vdash M : (\Pi x : A.B) \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]} \tag{8}$$

2. Специальное правило типизации:

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2} \tag{9}$$

где $s, s_1, s_2 \in K$

Вариации специального правила типизации.

Для $\lambda_{\rightarrow}$ возможен лишь один случай: $(\Pi x : A.B) : s \Rightarrow (\phi \rightarrow \psi)$.
1. В простой теории типов все типы имеют кайнд, состоящий из одной звездочки (случай зависимости терма от терма):

$$\frac{\Gamma, x : A \vdash M : B \;\; \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x.M) : (\Pi x : A.B)} \Rightarrow \frac{\Gamma, x : \phi \vdash M : \psi}{\Gamma \vdash (\lambda x.M) : \phi \rightarrow \psi} \tag{10}$$

$$\frac{\Gamma \vdash M : (\Pi x : A.B) \;\; \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]} \Rightarrow \frac{\Gamma \vdash M : \phi \rightarrow \psi \;\; \Gamma \vdash x : \phi}{\Gamma \vdash Mx : \psi} \tag{11}$$

То есть, функциональный тип является частным случаем зависимого произведения, таким случаем, где кайнд типов прост и тип области значений ψ не зависит от $x : \phi$.

Вариации специального правила типизации.

2. В полиморфном лямбда-исчислении несколько модифицированное правило (зависимость терма от типа):

$$\frac{\Gamma, \phi : * \vdash \psi : *}{\Gamma \vdash \forall \phi.\psi : *} \quad \frac{\Gamma \vdash M : \forall \phi.\psi \quad \Gamma \vdash \alpha : *}{\Gamma \vdash M\alpha : \psi[\phi := \alpha]} \tag{12}$$

3. Важнейший случай, который может рассматриваться, начиная с $\lambda\omega$ (зависимость типа от типа):

$$\frac{\Gamma \vdash s_1 : \Box \quad \Gamma \vdash s_2 : \Box}{\Gamma \vdash : s_1 \to s_2 : \Box} \quad \frac{\Gamma \vdash \phi : s_1 \to s_2 \quad \Gamma \vdash \psi : s_1}{\Gamma \vdash \phi\psi : s_2} \tag{13}$$

Вариации специального правила типизации.

4. И, наконец, зависимое произведение (зависимость типа от терма):
Рассмотрим два случая для правила применения, в первом случае $(\Pi x : A.B) : *$ у нас получается удаление П-типа:

$$\frac{\Gamma \vdash M : (\Pi x : A.B) \;\; \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]} \tag{14}$$

В другом, удаление П-кайнда:

$$\frac{\Gamma \vdash \psi : (\Pi x : \alpha.s) \;\; \Gamma \vdash N : \alpha}{\Gamma \vdash \phi N : s[x := N]} \tag{15}$$

Аналогично для абстракции:

$$\frac{\Gamma, x : \phi \vdash M : \phi \;\; \Gamma \vdash \psi : *}{\Gamma \vdash \lambda x.M : (\Pi x : \phi.\psi)} \;\; \frac{\Gamma, x : \phi \vdash M : \phi \;\; \Gamma \vdash \psi : k}{\Gamma \vdash \lambda x.\phi : (\Pi x : \phi.k)} \tag{16}$$

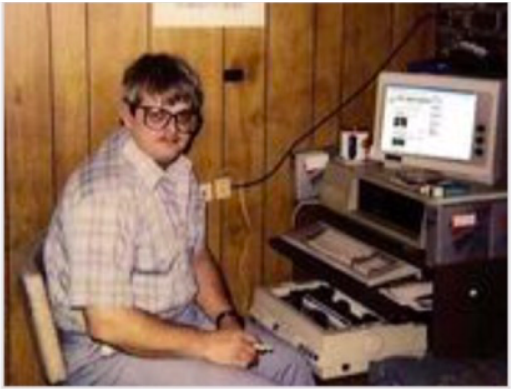
Язык Haskell

Определение языка согласно Lurkmore:

«Язык Haskell известен благодаря необходимости глубокого понимания лямбда-исчисления, замыканий, теории графов, теории категорий, сопромата, анатомии мозга и прочего матана даже для вывода на экран строки «Hello, World!»»

Язык Haskell

Текущий докладчик согласно тому же ресурсу



Типичный программист на Haskell

Типизация в языке Haskell и связанные с ней свойства языка.

1. Параметрический полиморфизм.
2. Специальный полиморфизм и классы типов.
3. Алгебраические типы данных.
4. Кайнды и контейнерные типы.
5. Больше классов типов: монады и функторы.

Параметрический полиморфизм.

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

curry :: ((a,b) -> c) -> a -> b -> c
curry = \f x y -> f (x, y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y

uniProd :: (a -> b) -> (a -> c) -> a -> (b, c)
uniProd = \f g x -> (f x, g x)

prod2 :: (a -> b) -> (d -> c) -> (a, d) -> (b, c)
prod2 = \f g p -> (f (fst p), g (snd p))
```

Специальный полиморфизм.

В случае специального полиморфизма функция может также вызываться на разных типах данных. Но каждый тип данных, используемый специальным полиморфизом, должен быть обеспечен реализацией текущего интерфейса.

Для этого используется конструкция класса типов, такой совокупности типов, каждый представитель которой удовлетворяет одному общему свойству.

В качестве простейшего примера рассмотрим реализацию сортировки Хоара (быстрая сортировка списка может быть реализована только в том случае, когда список состоит из элементов того типа, который является представителем класса типов `Ord`):

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (p:xs) = (quicksort (filter (< p) xs)) ++ [p] ++ (quicksort
(filter (>=) xs))
```

Реализация некоторого класса типов.

```
class Printable a where
 toString :: a -> String
 toString a = ['a']

instance Printable Bool where
 toString True = ''true''
 toString False = ''false''

instance Printable Int where
 toString n = show n
```

Алгебраические типы данных.

```
data TriangleABC = A0 | A1 | A2 deriving (Show, Eq, Ord)

angle :: TriangleABC -> Int
angle A0 = 0
angle A1 = 120
angle A2 = 240

infixr 4 'turn'

turn :: TriangleABC -> TriangleABC -> TriangleABC
turn x y | x == A0 and y == A0 = A0
 | (x == A0 and y == A1) || (y == A1 and x == A0) = A1
 | (x == A0 and y == A2) || (x == A2 and x == A0) = A2
 | x == A1 and y == A1 = A2
 | (x == A1 and y == A2) || (x == A2 and y == A1) = A0
 | x == A2 and y == A2 = A1
```

Алгебраические типы данных.

```
data List a = Nil | Cons a (List a) deriving Show

fromList :: List a -> [a]
fromList Nil = []
fromList (Cons x Nil) = [x]
fromList (Cons x xs) = x : fromList xs

toList :: [a] -> List a
toList [] = Nil
toList [x] = Cons x Nil
toList (x:xs) = Cons x (toList xs)
```

Кайнды и контейнерные типы.

Вполне реальная ситуация: не каждая вычислимая функция является всюду определенной. Есть необходимость упаковать значения функции в специальный контейнер, который, в зависимости от случая, либо возвращал бы значение функции, либо выводил специальное значение. Для таких случаев есть тип данных `Maybe`. Вызовем в `ghci` его кайнд:

```
Prelude> :k Maybe
Maybe :: * -> *
```

Наличие данного кайнда у типа `Maybe` говорит о том, что необходим типовой параметр, например, `Int` или `Bool`, чтобы получился тип данных, к которому можно непосредственно обращаться.

```
Prelude> :k Maybe Int
Maybe Int :: *
Prelude> :k Maybe Bool
Maybe Bool :: *
```

Кайнды и контейнерные типы.

Реализация данного типа в стандартной библиотеке языка:
`Maybe a = Nothing | Just a (deriving Show, Eq, Ord)`

Наиболее общая форма применения данного типа:

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing = n
maybe n f (Just x) = f x
```

Кайнды и контейнерные типы.

Для второго примера контейнерного типа есть смысл вспомнить схему аксиом интуиционистского исчисления высказываний:

$$(A \supset C) \supset ((B \supset C) \supset ((A \vee B) \supset C)) \tag{17}$$

Кейс такой: у нас есть две функции, у обеих функций одна кообласть, но области разные. Для этого мы можем реализовать функцию, выполняющую «разбор случаев». Для этого в стандартной библиотеке реализован контейнерный тип `Either`:

```
Prelude> :k Either
Either :: * -> * -> *
```

Реализация в стандартной библиотеке:

```
Either a b = Left a | Right b (deriving Show)
```

Кайнды и контейнерные типы.

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x) = f x
either f g (Right y) = g y
```

Применение типа `Either` в качестве реализации сообщения об ошибке (данный контейнерный тип можно использовать для обработки и вбрасывания исключений вручную, а также просьбы не делать интерпретатору грустно):

```
div :: Fractional a => a -> a -> Either String a
div x 0 = Left ''All men make faults, and even I in this, Authorizing thy
trespass with compare.''
div x y = Right (x/y)
```

Функторы.

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b

Prelude> :k Functor
Functor :: (* -> *) -> Constraint

instance Functor [] where
 fmap = map

instance Functor (Either a) where
 fmap f (Right x) = f x
 fmap f (Left x) = x
```

Законы функторов.

В построении экземпляра класса `Functor` необходимо следовать букве закона, нарушение которого чревато не очень хорошими последствиями:

1. `fmap id = id`
2. `fmap (f . g) = fmap f . fmap g`

Аппликативные функторы.

```
class Functor f => Applicative f where
 pure :: a -> f a
 (<*>) :: f (a -> b) -> f a -> f b

instance Applicative Maybe where
 pure = Just
 Nothing <*> _= Nothing
 (Just f) <*> something = fmap f something

Prelude> pure (+) <*> Just 3 <*> Just 8
Just 11
```

Моноиды.

```
class Monoid m where
 mempty :: m
 mappend :: m -> m -> m

instance Monoid [a] where
 mempty = []
 mappend = (++)
```

Язык Idris и его приятные отличия от Haskell.

Устанавливается через `cabal` (пакетный менеджер Хаскелла):
`cabal install Idris`

Отличительные (от Хаскелла) свойства языка:

0. Наличие типа типов.
1. Зависимая типизация.
2. Зависимое сопоставление с образцом.
3. Idris как прувер.

Тип типов в Idris.

В Idris определен тип Type, тип типов, модификация того, что в Хаскелле называлость кайндами:

```
Idris> :t Nat
Nat : Type
Idris> :t Bool
Bool : Type
Idris> :t List
List : Type -> Type
Idris> :t Either
Either : Type -> Type -> Type
Idris> :t Maybe
Maybe : Type -> Type
Idris> :t Either Int
Either Int : Type -> Type
```

Зависимые типы в Idris.

Объявим натуральные числа:

```
data Nat = Z | S Nat
```

Перепишем в синтаксисе наш Idris псевдокод с определением вектора через зависимую типизацию:

```
data Vect : Nat -> Type -> Type where
Nil : Vect Z a
(::) : a -> Vect k a -> Vect (S k) a
```

реализуем также конкатенацию векторов:

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil ys = ys
(x :: xs) ys = x :: xs ++ ys
```

Зависимые типы в Idris.

Зависимая типизация в Idris позволяет определить зависимые пары (Σ-типы, по Карри-Говарду, является обобщением экзистенциальной квантификации). В зависимых парах, в отличие от обычных кортежей, населяющих тип декартова произведения, вторая координата является зависимой от первой координаты:

```
data Sigma : (a : Type) -> (P : A -> Type) -> Type where
MkSigma : P : A -> Type -> (a : A) -> P a -> Sigma A P
```

Синтаксический сахар для `Sigma A (\y => Q)` — это `(a : A ** P)`, а `(x ** p)` — для `MkSigma x p`. Использование:

```
vec : (n : Nat ** Vect n Bool)
vec = (4 ** [True, False, True, False])
```

Зависимые типы в Idris.

Альтернативная реализация функции, конкатенирующей два вектора, с использованием зависимого паттерн-матчинга:

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++){n=Z} [] ys = ys
(++){n=S k} (x :: xs) ys = x :: xs ++ ys
```

Фильтрация вектора, данная функция возвращает вектор, содержащий те элементы, которые в исходным удовлетворяли некоторому предикату:

```
filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
filter [] p = (_ ** [])
filter p (x :: xs) with (filter p x)
 | (_ ** xs) = if (p x) then (_ ** x :: xs') else (_ ** xs')
```

Примеры простых доказательств.

```
data Or a b = inl a | inr b
data And a b = AndIntro a b

theorem1 : b -> Or a b
theorem1 x = inr x

theorem2 : a -> b -> And a b
theorem2 x y = AndIntro x y
```

Примеры простых доказательств.

```
data (=) : a -> b -> Type where refl : x = x

replace : {P : a -> Type} -> x = y -> P x -> P y

twoPlusTwo : 8 + 2 = 10
twoPlusTwo = refl
```

Спасибо за внимание!