

Generalized Algebraic Data Types (GADT)

Косарев Дмитрий а.к.а. Kakadu

матмех СПбГУ

7 ноября 2019 г.

В этих слайдах

1. Мотивация. Фантомные типы
2. GADT
3. Применение для интерпретатора мини-языка
4. Другие применения (кратко)
5. Равенство типов по Лейбницу
6. Теория про индексы де Брёйна (скорее всего не дойдем)

Алгебраические типы. Конструкторы данных

```
data ListInt =  
    Nil  
    | Cons Int ListInt
```

- **Cons** и **Nil** – конструкторы данных
- Это единственные способы построить значения типа **ListInt**
- Конструкторы по сути тэгируют пары значений
- И действуют как *динамические свидетели* эти значений

Функция `List.hd`

С точки зрения системы типов пустые и непустые списки *неразличимы*: и те, и другие имеют тип `ListInt`.

Но благодаря тегирующим конструкторам мы можем написать динамический тест, который локально определит природу списков:

```
hd :: ListInt -> Int
hd Nil          = {- List is empty          -} error "empty list"
hd (Cons x _)   = {- List is non-empty     -} x
```

Что на счет более безопасного получения головы?

Возможно ли изменить объявление типа списков так, чтобы получить вариант функции `hd`, которая всегда работает без ошибок?

Определение (Трюк программирования на уровне типов: фантомные типы)

Добавим дополнительный типовый параметр к объявлению типа, не встречающийся в этом определении типа, такой, что его можно свободно уточнить (*instantiate*), чтобы передать часть статической информации в систему проверки типов (*type checker*).

Другими словами, типовый параметр *a* значений *v* фантомного типа *T* а будет обозначать не тип какой-то компоненты *v*, а некоторую статическую информацию, привязанную к *v*.

Фантомные типы для свойства пустоты

Дополнительный типовый параметр

```
newtype Plist info = L ListInt
```

Свойство пустоты кодируется с помощью двух свежих различных типов

```
data Empty
```

```
data Nonempty
```

"Умные" конструкторы для конструирования данных

Превратим тип в абстрактный (скрыв его реализацию), а затем специализируем сигнатуру функций конструирования как нам нужно

```
nil :: Plist Empty
```

```
nil = L Nil
```

```
cons :: Int -> Plist b -> Plist Nonempty
```

```
cons x (L xs) = L (Cons x xs)
```

```
head :: Plist Nonempty -> Int
```

```
head (L (Cons x _)) = x
```

```
head (L Nil) = error "should not happen"
```


Более безопасное (?) извлечение головы списка

```
head :: Plist Nonempty -> Int
head (Cons x _ ) = x
head Nil          = error "should not happen"
```

Благодаря фантомным типам некоторые "плохие" вызовы `head` будут отклонены на стадии проверки типов.

Очевидно, что выражение `head Nil` не должно типизироваться, такт как типы `Empty` и `Nonempty` несовместимы.

Выражение же `head (Cons 1 Nil)`, наоборот, корректно типизируется.

```
head :: Plist Nonempty -> Int  
head (Cons x _ ) = x
```

Можем ли мы не разбирать второй случай, так как мы и так знаем что список не пустой?

Проблемка

```
head :: Plist Nonempty -> Int
head (Cons x _ ) = x
```

Можем ли мы не разбирать второй случай, так как мы и так знаем что список не пустой?

```
NonemptyList.hs:34:1-16: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘totalHd’: Patterns not matched: (Cons _ _)
|
34 | totalHd Nil = 42
    | ~~~~~
```

Компилятор недостаточно умен, чтобы доказать, что плохого случая не случится.

Механизм проверки типов для Обобщённых Алгебраических Типов Данных (GADTs) преодолевает упомянутое ограничение с помощью *автоматического уточнения контекста типизации в каждой ветке сопоставления с образцом*.

Первый GADT

```
data Empty
data Nonempty

data List prop where
  Nil  :: List Empty
  Cons :: Int -> List prop -> List Nonempty
```

Также как и "умные" конструкторы фантомных типов, это объявление *ограничивает типы значений, создаваемых с помощью конструкторов.*

К тому же, эти конкретные типы теперь привязаны к конструкторам данных, а не к "умным" конструкторам, что *позволяет типизировать сопоставление с образцом более точно.*

```
hd :: List Nonempty -> Int  
hd (Cons h _) = h
```

...и механизм проверки типов больше не будет жаловаться на то, что не все случаи были разобраны при сопоставлении с образцом.

Воистину, можно доказать, что не упомянутый случай, относящийся к конструктору **Nil** невозможен, так как такой паттерн может использоваться только со значениями типа **List Empty**, которые несовместны с типом **List Nonempty**.

Ещё пример

```
hd :: List Nonempty -> Int
```

```
totalHd :: List a -> Int
```

```
totalHd Nil = 42
```

```
totalHd xs@(Cons _ _) = hd xs
```

Во второй ветки компилятор смог передать значение `xs` типа `List` а в функцию, которая ожидала тип `List Nonempty`.

Равенство типов

Мы можем переформулировать предыдущее определение GADT путём добавления равенств типов к каждому конструктору данных:

```
data List prop where
  Nil  :: prop ≡ Empty    → List prop
  Cons :: prop ≡ Nonempty → Int -> List prop -> List prop
```

Внимание: воображаемый¹ синтаксис!

¹Хотя что-то подобное в Haskell есть

В правой части каждой ветки паттерн-мэтчинга обрабатывается отдельный конструктор и равенства типов *неявно* добавляются в контекст Γ , в котором находится type checker.

```
totalHd :: List a -> Int
totalHd Nil = 42
totalHd xs@(Cons _ _) = hd xs
```

Вывод типов для второй ветки работает примерно так:

- Мы знаем что $a \equiv \text{Nonempty}$
- А также в контексте Γ написано, что xs имеет тип List Nonempty :
 $\Gamma \vdash xs :: \text{List Nonempty}$
- Из того, что a — это Nonempty , по смыслу типов следует, что $\text{List } a$ — это List Nonempty :
 $a = \text{Nonempty} \vdash \text{List } a = \text{List Nonempty}$
- Подставив типы, получим:
 $\Gamma \vdash xs :: \text{List Nonempty}$

Упражнения (1/3)

Абстрактные типы для натуральных чисел в стиле Пеано: "ноль" и "следующий".

```
type Zero
```

```
type Succ a
```

Используя эти типы напишите другую реализацию типа `IntList`, которая позволит хранить на уровне типов информацию о длине списка.

Упражнения (2/3)

Рассмотрим функцию попарного суммирования списков:

```
sum :: [Int] -> [Int] -> [Int]
sum [] [] = []
sum (x:xs) (y:ys) = (x+y):(sum xs ys)
sum _ _ = error "different lengths"
```

Перепишите её с помощью списка из упражнения выше. Получился ли более точный тип по сравнению с предыдущей реализацией?

Упражнения (3/3)

Рассмотрим функцию слияния списков:

```
append :: [Int] -> [Int] -> [Int]
append [] xs = xs
append (x:xs) ys = x:(append xs ys)
```

Можно ли с помощью только что введенных списков получить правильно типизированную реализацию?

Язык выражений с парами, проекциями и числами

```
-- t ::= 0, 1, ... /  $\pi_1$  t /  $\pi_2$  t / (t, t)
```

```
data Term =
```

```
    Const Int
```

```
  | Pair Term Term
```

```
  | Fst Term
```

```
  | Snd Term
```

```
data Value = VInt Int | VPair Value Value
```

```
eval :: Term -> Value
```

Интерпретатор "в лоб"

```
eval :: Term -> Value
eval (Const n) = VInt n
eval (Pair l r) = VPair (eval l) (eval r)
eval (Fst t) = case eval t of
    VInt _ -> error "only pairs can be projected"
    VPair a _ -> a

eval (Snd t) = ...
```

Как убедить себя, что если выражение мини-языка построено правильно, то интерпретатор не упадет?

Интерпретатор выражений с комментариями (1/3)

*{- eval переводит правильно построенные выражения типа T в значения типа T,
или более формально:*

$\forall e \in T. Pre : " \vdash e : T " \Rightarrow Pre : " \vdash eval\ e : T "$

-}

eval :: **Term** -> **Value**

eval (**Const** n) = **VInt** n

-- Путём разбора случаев (inversion) в Pre получим, что $\vdash Const\ n : int$

-- Следовательно, $T = int$, и это то, что нужно: $VInt\ x : Int$

eval (**Pair** t1 t2) = **VPair** (eval t1) (eval t2)

-- Путём разбора случаев (inversion) в Pre узнаём, что существуют β_1 и β_2

-- такие, что $\vdash Pair(t_1, t_2) : (\beta_1, \beta_2)$, $\vdash t_1 : \beta_1$ и $\vdash t_2 : \beta_2$.

-- Следовательно, $\vdash eval\ t_1 : \beta_1$ и $\vdash eval\ t_2 : \beta_2$

-- Итого, получаем, что $\vdash (eval\ t_1, eval\ t_2) : (\beta_1, \beta_2)$

eval (**Fst** t) = ...

eval (**Snd** t) = ...

Интерпретатор выражений с комментариями (2/3)

{- eval е переводит правильно построенные выражения типа T в значения типа T, или более формально:

$\forall e \in T. Pre : " \vdash e : T " \Rightarrow Pre : " \vdash eval\ e : T "$

-}

```
eval :: Term -> Value
```

```
eval (Fst t) = case eval t of
```

```
  VInt _ -> error "only pairs can be projected"
```

```
  VPair v1 _ -> v1
```

-- Путём разбора случаев (inversion) в Pre узнаём, что

-- $\exists \beta \vdash t : (T, \beta)$

-- Затем, $\vdash eval\ t : (T, \beta)$

-- Т.к. результат имеет тип пары, то $eval\ t = (v_1, v_2)$, где $v_1 : T$

```
eval (Snd t) = ...
```


Интерпретатор выражений с комментариями (3/3)

```
eval (Fst t) = case eval t of
  VInt _ -> error "only pairs can be projected"
  VPair v1 _ -> v1
  -- Итого получаем, что  $eval\ t = (v_1, v_2)$ 
eval (Snd t) = ...
```

Отсюда следует, что мы разбираем всегда пару, а вторая ветка паттерн-мэтчинга никогда не случится, они лишняя (redundant).

Как закодировать это свойство используя типы?

Тип интерпретатора может быть обогащен типовой переменной, которая обозначает результат интерпретации.

```
eval :: Term a -> Value a
```

В данном конкретном случае конструкторы типов **Term** и **Value** выступают в роли фантомных типов. Вопрос: в нашем случае какую именно информацию они кодируют?

Кодирование предикатов, используя типы

Что означает $e : \text{Term } a$?

Выражение e имеет тип a .

Другими словами, мы кодируем на уровне типов предикат, который обозначает "правильную типизированность" нашего языка выражений. Чтобы всё сделать правильно, нам нужно ещё закодировать типы в базовом (host) языке программирования. Это может быть сделано с использованием конструкторов типа:

```
--  $\tau ::= \text{int} \mid (\tau, \tau)$   
data IntType  
data PairType a b
```

Язык корректно построенных выражений

```
-- t ::= 0, 1, ... /  $\pi_1$  t /  $\pi_2$  t / (t, t)
data Term a where
  Lit    :: Int          -> Term IntType
  Pair   :: Term a -> Term b    -> Term (PairType a b)
  Fst    :: Term (PairType a b) -> Term a
  Snd    :: Term (PairType a b) -> Term b

-- v ::= 0, 1, ... / (v, v)
data Value a where
  VInt    :: Int -> Value IntType
  VPair   :: Value a -> Value b -> Value (PairType a b)
```

Здесь объявляются два GADTs: **Term** и **Value**.

Интерпретатор корректно типизированных выражений

```
eval :: Term a -> Value a
eval (Const n) =
    -- a == IntType
    VInt n
eval (Pair t1 t2) =
    --  $\exists b\ c. a = \text{PairType } b\ c, t1 : \text{Term } b, t2 : \text{Term } c$ 
    VPair (eval t1, eval t2)
eval (Fst t) =
    --  $\exists b. t : \text{Term } (\text{PairType } a\ b)$ 
    case eval t of
        VPair v1 _ -> v1
eval (Snd t) =
    --  $\exists b. t : \text{Term } (\text{PairType } b\ a)$ 
    case eval t of
        VPair _ v2 -> v2
```

Эквивалентность типов с учетом равенств

Используя стандартную процедуру вывода типов, мы можем получить, что во второй ветке

$$\text{VPair (eval t1) (eval t2)} \quad :: \quad \text{Value (PairType c d)}$$

Это синтаксически отличается от ожидаемого типа **Value a**.

К счастью, в этой ветке локально присутствует информация о равенстве типов **a = PairType c d**.

Из этого системы проверки типов может доказать, что:

$$\text{a = PairType c d} \quad \models \quad \text{Value a = Value (PairType c d)}$$

Что очевидно верно в системе типов языка Haskell (и системах на основе алгоритма вывода типов Хиндли-Милнера).

```
eval :: Term a -> Value a
eval (Const n) = ...
eval (Pair t1 t2) = ...
eval (Fst t) =
  --  $\exists b. t : \text{Term } (\text{PairType } a \ b)$ 
  case eval t of
    -- Единственный паттерн, который тут может быть,
    -- должен иметь тип PairType a b
    VPair v1 _ -> v1
eval (Snd t) = ...
```

Tagless интерпретатор

Во всех вложенных pattern matching у нас стоят одиночные паттерны, которые покрывают все возможные случаи значений. В некотором смысле они "неопровержимые" (irrefutable).

Это означает, что мы можем безопасно объявить `type Value a = a`

Другими словами, мы можем использовать базовый (host) язык с GADT (в этих слайдах это Haskell) одновременно и как язык для написания интерпретатора, и как язык, выражает значения и типы встраиваемого языка.

Язык корректно типизированных выражений

С учетом нового определения типа **Value** мы получаем новое определение GADT типа **Term**

```
-- t ::= 0, 1, ... /  $\pi_1$  /  $\pi_2$  / (t, t)
data Term a where
  Const :: Int -> Term Int
  Pair   :: Term a -> Term b -> Term (a,b)
  Fst    :: Term (a,b) -> Term a
  Snd    :: Term (a,b) -> Term b

-- v ::= 0, 1, ... / (v, v)
type Value a = a
```

Сейчас мы не только избавились от конструкторов, подчистив код, но также его линейно ускорив [5].

Окончательная реализация tagless интерпретатора

```
eval :: Term a -> Value a
eval (Const n) =
  -- a = Int
  n
eval (Pair l r) =
  --  $\exists b\ c. a = (b, c), t1 : Term\ b, t2 : Term\ c$ 
  (eval l, eval r)
eval (Fst t) =
  --  $\exists b. t : Term\ (a, b)$ 
  fst (eval t)
eval (Snd t) =
  --  $\exists b. t : Term\ (b, a)$ 
  snd (eval t)
```

Другие применения GADT

- Красно-черные деревья, сбалансированные по построению [3][5].
- Или **то же самое** для 2-3 деревьев.
- Simply-typed Lambda Calculus интерпретатор [2], где используется GADT для корректного (каждое имя переменной вводилось λ -абстракцией) представления выражений.
- Можно сделать из GADT типизированное внутреннее представление программы [6].
- Эмуляция некоторых свойств зависимых типов, когда в самом языке зависимых типов нет.
- Различное представление в памяти различных данных [7].

Равенство типов по Лейбницу

Определение (Равенство по Лейбницу)

Две сущности равны, если они неразличимы (а, следовательно – взаимозаменяемы) в любом контексте.

Равенство типов по Лейбницу

Определение (Равенство по Лейбницу)

Две сущности равны, если они неразличимы (а, следовательно – взаимозаменяемы) в любом контексте.

Определение типа равенства с помощью GADT:

```
data Eq a b where  
  Refl :: Eq a a
```

Другое определение типа равенства на случай, если GADT в языке нет, но есть экзистенциальные типы и типы высшего порядка (higher-kinded types):

```
data Eq a b = forall f . (f a -> f b, f b -> f a)
```

Легко понять, то если существует типобезопасная функция с таким типом, то она всегда зависит

```
test1 :: a -> (b -> c) -> c  
test1 = error "doesn't exist"
```

Легко понять, то если существует типобезопасная функция с таким типом, то она всегда зависит

```
test1 :: a -> (b -> c) -> c
test1 = error "doesn't exist"
```

Но мы можем это попробовать исправить, явно передав информацию, что два типа равны.

```
data Eq a b where
  Refl :: Eq a a

test2 :: Eq a b -> a -> (b -> c) -> c
test2 Refl x f = f x
```

Тип `Eq` – это отношение равенства

- Рефлексивность

```
Refl :: Eq a a
```

- Симметричность

```
symm :: Eq a b -> Eq b a
```

```
symm Refl = Refl
```

- Транзитивность

```
trans :: Eq a b -> Eq b c -> Eq a c
```

```
trans Refl Refl = Refl
```

Если у вас нет в языке GADT (например, вы пишете на F#), то с помощью обычных алгебраических типов и `Eq`, вы можете смулировать наличие GADT.

Дальше есть слайды про ещё одно применение,
но на них времени скорее всего не хватит.

Безымянное представление через индексы де Брёйна (de Bruijn)

Идея

- Заводим глобальный контекст Γ , где взаимно однозначно сопоставляем каждому натуральному числу имя переменной.
- Связанные переменные представляем числом $k > 0$. Оно означает, что переменная связывается k -й охватывающей лямбдой.
- Свободная переменная x представляются в виде суммы Γx и глубины её местоположения внутри терма в λ абстракциях.

Пример: $\Gamma = \{b \mapsto 0, a \mapsto 1, z \mapsto 2, y \mapsto 3, x \mapsto 4\}$

- $x(yz) \equiv 4(3\ 2)$
- $(\lambda w \rightarrow yw) \equiv (\lambda \rightarrow 4\ 0)$
- $(\lambda w \rightarrow yx) \equiv (\lambda \rightarrow \lambda \rightarrow 6)$

Подстановка в безымянном представлении $[k \mapsto s]t$ (1/2)

Пример: $[1 \mapsto s](\lambda \rightarrow 2) = [x \mapsto s](\lambda y \rightarrow x)$

Когда s проникнет под абстракцию, то надо будет "сдвинуть" некоторые индексы переменных, но не все, например, если $s = 2(\lambda \rightarrow 0)$ (т.е. $s = z(\lambda w \rightarrow w)$), то надо сдвинуть 2, а не 0.

Определение

Сдвиг терма t на d позиций с отсечкой c (обозначается $\uparrow_c^d(t)$)

- $\uparrow_c^d(k) = \begin{cases} k, & \text{если } k < c \\ k + d, & \text{если } k \geq c \end{cases}$
- $\uparrow_c^d(\lambda \rightarrow t_1) = \lambda \rightarrow \uparrow_{1+c}^d(t_1)$
- $\uparrow_c^d(t_1 t_2) = \uparrow_c^d(t_1) \uparrow_c^d(t_2)$

Подстановка в безымянном представлении $[k \mapsto s]t$ (2/2)

Определение

Подстановка терма s вместо переменной номер j (обозначается $[j \mapsto s]t$)

- $[j \mapsto s]k = \begin{cases} s, & \text{если } k = j \\ k, & \text{в противном случае} \end{cases}$
- $[j \mapsto s](\lambda \rightarrow t_1) = (\lambda \rightarrow [(j+1) \mapsto \uparrow_0^1 s]t_1)$
- $[j \mapsto s](t_1 t_2) = ([j \mapsto s]t_1 [j \mapsto s]t_2)$

Упражнения на подстановку с индексами де Брёйна

- $[b \mapsto a](b(\lambda x \rightarrow \lambda y \rightarrow b))$
- $[b \mapsto a(\lambda z \rightarrow a)](b(\lambda x \rightarrow b))$
- $[b \mapsto a](\lambda b \rightarrow b \ a)$
- $[b \mapsto a](\lambda a \rightarrow b \ a)$

-  Демки на Haskell
[Gitlab repo](#)
-  Glamorous Glambda interpreter
Richard Eisenberg
[github repo](#)
[YouTube Video](#)
-  Red-Black trees, balanced by construction
[Github gist](#)
-  GADT slides in OCaml context
Yann Régis-Gianas
[PDF slides](#)

-  Investigation of GADT applications and usage
Parth Shah
[PDF](#)
-  A Type-Preserving Compiler in Haskell
Louis-Julien Guillemette & Stefan Monnier
[PDF](#)
-  Why GADTs matter for performance
Yaron Minsky
[blog post](#)