Generalized Algebraic Data Types (GADT)

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

8 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 имею 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?

Фантомные типы

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

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

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

Проблемка

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

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

```
NonemptyList.hs:30:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘head’: Patterns not matched: (L Nil)
```

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

Generalized Algebraic Data Types

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

```haskell
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.*

```haskell
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 a` 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: воображаемый[1] синтаксис!

[1] Хотя 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` $\equiv$ `Nonempty`
- А также в контексте Γ написано, что `xs` имеет тип `List Nonempty`:
  $\Gamma \vdash$ `xs :: List Nonempty`
- Из того, что `a` — это `Nonempty`, по смыслу типов следует, что `List a` — это `List Nonempty`:
  `a = Nonempty` $\vDash$ `List a = List Nonempty`
- Подставив типы, получим:
  $\Gamma \vdash$ `xs :: List Nonempty`

Упражнения (1/3)

Астрактные 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 списком из упражнения выше. Получился ли более точный тип по сравнению с предыдущей реализацией?

Упражнения (3/3)

Рассмотрим функцию слияния списков:

```haskell
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

```haskell
-- t ::= 0, 1, ... | π₁ t | π₂ 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 e переводит правильно построенные выражения типа T в значения типа T,
   или более формально:
   ∀ e T . Pre : "⊢ e : T" ⇒ Post : "⊢ eval e : T"
-}
eval :: Term -> Value
eval (Const n) = VInt n
  -- Путём разбора случаев (inversion) в Pre получим, что ⊢ 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 β₁ и β₂
  -- такие, что ⊢ Pair(t₁,t₂) : (β₁,β₂), ⊢ t₁ : β₁ и ⊢ t₂ : β₂.
  -- Следовательно, ⊢ eval t₁ : β₁ и ⊢ eval t₁ : β₁
  -- Итого, получаем, что ⊢ (eval t₁, eval t₂) : (β₁, β₂)
eval (Fst t) = ...
eval (Snd t) = ...
```

Интерпретатор выражений с комментариями (2/3)

```
{- eval e переводит правильно построенные выражения типа T в значения типа T,
   или более формально:
   ∀ e T . Pre : "⊢ e : T" ⇒ Post : "⊢ 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
  -- ∃β ⊢ t : (T, β)
  -- Затем, ⊢ eval t : (T, β)
  -- Т.к. результат имеет тип пары, то eval t = (v₁, v₂), где v₁ : 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 = (v1, v2)
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 : 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 использование конструкторов типа:

```
-- τ ::= int | (τ,τ)
data IntType
data PairType a b
```

Язык корректно построенных выражений

```haskell
-- t ::= 0, 1, ... | π₁ t | π₂ t | (t, t)
data Term a where
  Const :: 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) =
  -- ∃ b c. a = PairType b c, t1 : Term b, t2 : Term c
  VPair (eval t1, eval t2)
eval (Fst t) =
  -- ∃ b. t : Term (PairType a b)
  case eval t of
    VPair v1 _ -> v1
eval (Snd t) =
  -- ∃ b. t : Term (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

```
VPair (eval t1) (eval t2)  ::  Value (PairType c d)
```

Это синтаксически отличается от ожидаемого типа `Value a`.

К счастью, в этой ветке локально присутсвует информация о равенстве типо
`a = PairType c d`.

Из этого системы проверки типов может доказать, что:

```
a = PairType c d  ⊨  Value a = Value (PairType c d)
```

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

```haskell
eval :: Term a -> Value a
eval (Const n) = ...
eval (Pair t1 t2) = ...
eval (Fst t) =
  -- ∃ b.  t : Term (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 па́ттерны, 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, который выражает значения и типы встраиваемого языка.

Язык корректно типизированных выражений

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

```haskell
-- t ::= 0,1,... | π1 | π2 | (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

```haskell
eval :: Term a -> Value a
eval (Const n) =
  -- a = Int
  n
eval (Pair l r) =
  -- ∃ b c. a = (b,c), t1 : Term b, t2 : Term c
  (eval l, eval r)
eval (Fst t) =
  -- ∃ b. t : Term (a, b)
  fst (eval t)
eval (Snd t) =
  -- ∃ 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 зависает

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

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

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

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

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

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

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

Определение

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

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

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

Определение

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

- $[j \mapsto s]k = \begin{cases} s, & \text{если } k = j \\ k, & \text{в противном случае} \end{cases}$
- $[j \mapsto s](\lambda \to t_1) = (\lambda \to [(j+1) \mapsto \uparrow^1_0 s]t_1)$
- $[j \mapsto s](t_1 t_2) = ([j \mapsto s]t_1 [j \mapsto s]t_2)$

Упражнения на подстановку с индексами де Брёйна

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

Ссылки I

- 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

Ссылки II

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