GADT & DSL

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

29 ноября 2018 г.

Эквивалентность типов

```
type EQUIV a b = (a -> b, b -> a)
```

Эквивалентность означает, что мы всегда может преобразовать одно в другое.
Доказательством будет пара функций f и g, такая что $f \cdot g \equiv g \cdot f \equiv id$

Т.е. если у нас есть `(f,g)` – доказательсво `EQUIV a Int`, а также `x::a`, то мы можем сковертировать `a` в `Int`.

Свойства эквивалентности

- Рефлексивность
  `refl :: EQUIV a a`
- Симметричность
  `sym :: EQUIV a b -> EQUIV b a`
- Транзитивность
  `sym :: EQUIV a b -> EQUIV b c -> EQUIV a c`

Но есть кое-что более сильное чем эквивалентность: равенство.

Равенство типов

(По Лейбницу) Типы равны если, и только если их можно заменять друг на друга в любых контекстах.

```
type EQUAL a b = forall c . c a -> c b
```

Слово `forall` указывает на то, что тип *экзистенциальный* (existential).

Типовую переменную `c` не надо указывать в списке типовых параметров: она скрыта квантором.

Почему такое название станет ясно позднее.

Примеры

Типы `(Int, Bool)` и `(Bool, Int)` эквивалентны, так как изоморфны, но не равны.

Пара функций `swap` – это искомая пара изоморфизмов:
`(swap,swap) :: EQUIV (Int, Bool) (Bool,Int)`.

Но не любые функции, что населяет тип – изоморфизм. Например,
`(const 0, const 1) :: EQUIV Int Int` – не изоморфизм.

Примеры

Типы `(Int, Bool)` и `(Bool, Int)` эквивалентны, так как изоморфны, но не равны.

Пара функций `swap` – это искомая пара изоморфизмов: `(swap,swap) :: EQUIV (Int, Bool) (Bool,Int)`.

Но не любые функции, что населяет тип – изоморфизм. Например, `(const 0, const 1) :: EQUIV Int Int` – не изоморфизм.

Для равенства всё получше: не существует завершающихся жителей `EQUAL (Int, Bool) (Bool,Int)`. Если бы он существовал, то он должен был бы уметь делать замену в произвольном контексте.

Более того, в `EQUAL Int Int` живет только `id`: в произвольном контексте нет другой алтернативы кроме как вернуть свой аргумент.

Равенство + алгебраические типы

```
data Foo a = AAA (EQUAL a Int)    a
           | BBB (EQUAL a String) a
```

Типобезопасно по построению

```
Prelude> :t AAA id
AAA id :: Int -> Foo Int
Prelude> :t BBB id
BBB id :: String -> Foo String
```

Равенство + алгебраические типы

```
data Foo a = AAA (EQUAL a Int)    a
           | BBB (EQUAL a String) a
```

Типобезопасно по построению

```
Prelude> :t AAA id
AAA id :: Int -> Foo Int
Prelude> :t BBB id
BBB id :: String -> Foo String
```

Данные можно (немного костыльно) преобразовывать.

```
cast :: EQUAL a b -> a -> b
cast _ = Unsafe.Coerce.unsafeCoerce
```

GADT

Generalized Algebraic Data Types сделали экзистенциальную квантификацию устаревшей.

Пример:

```
{-# LANGUAGE GADTs #-}

data Foo a where
  AAA :: Int    -> Foo Int
  BBB :: String -> Foo String
```

Имеем функции-конструкторы для типа данных

```
AAA :: Int    -> Foo Int
BBB :: String -> Foo String
```

GADT. Пример.

Без типовой аннотации – ошибка компиляции: не могу сунифицировать Int и String.

```
foo (B x) = x
foo (A x) = x
```

С типовой аннотацией – OK

```
foo :: Foo a -> a
foo (B x) = x
foo (A x) = x
```

Почему называются экзистенциальные – рассказывать здесь.

DSL = Domain Specific Language

т.е. язык, специфичный для конкретной предметной области.

Примеры предметных областей: синтаксический анализ, запросы к базам данных, нахождение/фильтрация спама в соц.сети, и т.д.

Плюсы DSL – они лучше отражают предметную область.

Примеры DSL: SQL, dot для задания графов, парсер-комбинаторы, синтаксический сахар над LINQ в C#, Doxygen.

Embedded DSL

Embedded DSL – встраивание языка, специфичного для предметной области, в язык общего назначения.

Примеры EDSL: парсер-комбинаторы, синтаксический сахар над LINQ в C#, Doxygen, printf (!?), и т.д.

Встраивание бывает разных видов:

- Встраивание непосредственно в язык vs. просто рядом (как Doxygen).
- EDSL поддержанный компилятором vs. объявленные программистом функции
- Сильно типизированный DSL vs. слабо типизированный
- Встраивание только синтаксиса (deep embedding) vs. и семантики тоже (shallow embedding).

Упражнение

Попробуйте спроектировать типы мини-языка и сделать функции Haskell, так, чтобы код с использованием этих функций был более-менее похож на оригинальный язык. Например:

- a lá SQL:
```
select all from table1
where (\o -> name e = "OK")
```
- a lá printf: `printf (lit "hello, "^^ string) "world!"`
- Что-то из финансовой области

```
andGive :: Contract -> Contract -> Contract
andGive c d = c `and` (give d)
```

Начальное (initial) встраивание

```
data Exp = Lit Int          -- constant
         | Neg Exp          -- unary negation
         | Add Exp Exp      -- sum

ti1 = Add (Lit 8)  (Neg (Add (Lit 1)  (Lit 2)

eval :: Exp -> Int
eval   (Lit n)      = n
eval   (Neg e)      = - (eval e)
eval   (Add e1 e2) = eval e1 + eval e2
```

Другой способ (final)

Если нас интересует только результат `eval`'а

```
type Repr = Int

lit :: Int -> Repr
lit n = n

neg :: Repr -> Repr
neg e = - e

add :: Repr -> Repr -> Repr
add e1  e2 = e1 + e2

tf1 :: Repr
tf1 = add (lit 8) (neg (add (lit 1) (lit 2)))
```

Сравним два способа

Initial

- Отличия между `tf1` и `ti1` только в регистре буковок
- Pattern matching понятно как делать.

Final

- 👍 *Композициональность* – сделать сложение можно в отрыве, имея только результаты левой и правой частей, не задумываясь в каком контексте используется сумма.
- Pattern matching не очень понятно как делать.
- 👎 Менее общий способ?
- 👍 Или нет?

Show для initial

```
view :: Exp -> String
view  (Lit n)      = show n
view  (Neg e)      = "( - " ++ view e ++ ")"
view  (Add e1 e2) = "(" ++ view e1 ++ " + " ++
                           view e2 ++ ")"
```

А вот в final у нас тип `data Repr = Int` зафиксирован, и мы не можем написать преобразование в строку.
Вот если бы поведение функции могло зависеть от типа....

Symantics = Syntax + Semantics

```
class ExpSYM repr where
  lit :: Int  -> repr
  neg :: repr -> repr
  add :: repr -> repr -> repr

tf1 :: ExpSYM repr => repr
tf1 = add (lit 8) (neg (add (lit 1) (lit 2)))
```

Данные описываются не только синтаксической структурой, но также семантическим смыслом, которые придали этим данным

Eval для ExpSYM

```
instance ExpSYM Int where
  lit n = n
  neg e = -e
  add e1 e2 = e1 + e2

eval :: Int -> Int
eval = id
```

`eval` теперь ничего не считает, не делает pattern matching, следовательно он пошустрее, чем в initial виде.

Show для ExpSYM

А ещё мы можем сделать pretty-printer

```
instance ExpSYM String where
  lit n      = show n
  neg e      = "-(" ++ e ++ ")"
  add e1 e2 = "(" ++ e1 ++ "+" ++ e2 ++ ")"

eval :: String -> String
eval = id
```

Задача про pattern matching

У нас есть язык, где есть две бинарные операции, переменные и унарная операция.

Например, числа со сложением, умножением и унарной операцией "минус". Или формулы логики высказываний с конъюнкцией, дизъюнкцией и отрицанием формул.

Требуется преобразовывать формулы в особую формулу, которая выглядит сумма произведений (или конъюнкция дизъюнкций) литералов, где литерал – это или переменная, или её отрицание.

Сделайте вначале в initial, потом в final стиле.

Expression problem

"The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)."

Phillip Wadler

Оригинальный e-mail с примером кода на "Generic" Java (1998)
Разные методы решения expression problem находятся здесь.

Расширяем язык

```
data Exp = Lit Int
         ...
         | Mul Exp Exp  -- product


eval (Mul a b) = ...
eval ...
```

Расширяем final вариант

```
class MulSYM repr where
  mul :: repr -> repr -> repr

tfm2 :: (ExpSYM repr, MulSYM repr) => repr
tfm2 = mul (lit 7) tf1
```

```
instance MulSYM Int where
  mul e1 e2 = e1 * e2
instance MulSYM String where
  mul e1 e2 = "(" ++ e1 ++ " * " ++ e2 ++ ")"
```

Некоторые выводы

- Легко добавлять новые интерпретации языка
- Но также и новые формы (т.е. интерпретаторы расширяемы по-умолчанию)
- Типовая безопасность: если забыть `instance MulSYM String` – не скомпилируется

Пример про сериализацию

Сериализовывать будет в простое дерево, хотя можно и в JSON

```
data Tree = Leaf String
          | Node String [ Tree]
   deriving (Eq, Read, Show)


instance ExpSYM Tree where
  lit n      = Node "Lit" [Leaf (show n)]
  neg e      = Node "Neg" [e]
  add e1 e2 = Node "Add" [e1,e2]

toTree :: Tree -> Tree
toTree   = id
tf1 tree = toTree tf1
```

Сериализация. Пример.

Сериализовывать будет в простое дерево, хотя можно и в JSON

```
instance ExpSYM Tree where
  lit n     = Node "Lit" [Leaf (show n)]
  neg e     = Node "Neg" [e]
  add e1 e2 = Node "Add" [e1,e2]

tf1 :: ExpSYM repr => repr
tf1 = add (lit 8) (neg (add (lit 1) (lit 2)))

{-
Node "Add"
     [Node "Lit" [Leaf "8"],
      Node "Neg" [Node "Add"
                       [Node "Lit" [Leaf "1"],
                        Node "Lit" [ Leaf "2"]]]] -}
```

А теперь – десериализация

Сериализовывать будет в простое дерево, хотя можно и в JSON

```
data Tree = Leaf String
          | Node String [ Tree]
   deriving (Eq, Read, Show)


instance ExpSYM Tree where
  lit n      = Node "Lit" [Leaf (show n)]
  neg e      = Node "Neg" [e]
  add e1 e2 = Node "Add" [e1,e2]

toTree :: Tree -> Tree
toTree   = id
tf1 tree = toTree tf1
```

Базовый случай десериализации

Десериализация – частичная функция

```
type ErrMsg = String

safeRead :: Read a => String -> Either ErrMsg a
safeRead s = case reads s of
  [( x, "")] -> Right x
  _ -> Left ("Read error: " ++ s)
```

Десериализация целиком

```
fromTree (Node "Lit" [Leaf n]) =
                          liftM lit (safeRead n)

fromTree (Node "Neg" [e]) = liftM neg (fromTree e)

fromTree (Node "Add" [e1,e2])   =
          liftM2 add (fromTree e1) (fromTree e2)
fromTree e  = Left ("Invalid tree : " ++ show e)
```

где

```
liftM  :: Monad m => (a1 -> r) -> m a1 -> m r
liftM2 :: Monad m => (a1 -> a2 -> r) ->
             m a1 -> m a2 -> m r
```

Воспользуемся десериализованным

```
test eval =
  case fromTree tf1 tree of
    Left e  -> putStrLn ("Error: " ++ e)
    Right x -> print (eval x)
```

Упражнение: воспользуйтесь в `test` и функцией `eval`, и функцией `show`. Решите проблемы с типами, которые случатся.

Конец.

Ссылки I

- Oleg Kiselyov.
  Typed Tagless Final Interpreters.
  ссылка.
- Simon Peyton Jones and J-M. Eber.
  How to write financial contract.
  ссылка.
- Ralf Lämmel and Oleg Kiselyov.
  Exploring typed language design in Haskell.
  ссылка.
- Jeremy Gibbons
  Folding Domain-Specific Languages: Deep and Shallow Embeddings.
  ссылка.