

# CS314. Функциональное программирование

## Лекция 18. Метапрограммирование средствами Template Haskell

Направление «Фундаментальная информатика и информационные технологии»  
Институт математики, механики и компьютерных наук имени И. И. Воровича  
Южный федеральный университет

8 декабря 2017 г.

- 1 Идея метапрограммирования и Template Haskell
- 2 Пример: генерация функций-проекторов для кортежей
- 3 Реификация и квазицитирование

# Содержание

- 1 **Идея метапрограммирования и Template Haskell**
- 2 Пример: генерация функций-проекторов для кортежей
- 3 Реификация и квазицитирование

# Идея метапрограммирования

- Метапрограммирование — **writing code that writes code**.
- Примеры средств метапрограммирования: макросы языка Лисп, C++ Templates, Template Haskell.
- Примеры применения: генерация очевидного кода, предварительные вычисления, обход языковых ограничений.
- Всегда можно сгенерировать текст исходного файла на языке, приспособленном для работы с текстом (Perl!), а потом его скомпилировать.

# Template Haskell: цитаты и монада Q

```
$ ghci -XTemplateHaskell
GHCi, version 7.10.3: http://www.haskell.org/ghc/ :? for help
ghci> :m +Language.Haskell.TH
ghci> runQ [| 1+2 |]
InfixE (Just (LitE (IntegerL 1)))
        (VarE GHC.Num.+)
        (Just (LitE (IntegerL 2)))
```

- [| ... |] — оксфордские скобки (цитаты).
- runQ — запуск вычисления в монаде Q.
- Доступ к абстрактному синтаксическому дереву (AST).
- Возможность создавать и перестраивать AST.

## Template Haskell: внедрение (splicing) кода

```
ghci> $( return (LitE (IntegerL 42)) )  
42
```

```
ghci> $( return (InfixE (Just (LitE (IntegerL 1)))  
                        (VarE (mkName "+"))  
                        (Just (LitE (IntegerL 2)))) ) )  
3
```

# Код Template Haskell в исходном коде

- Важное ограничение: внедряемые определения TH должны импортироваться из других модулей.

## Hello.hs

```
{-# LANGUAGE TemplateHaskell #-}  
module Hello where  
import Language.Haskell.TH  
  
hello :: Q Exp  
hello = [| putStrLn "Hello world" |]
```

## useHello.hs

```
{-# LANGUAGE TemplateHaskell #-}  
module Main where  
import Hello  
main = $(hello)
```

# Содержание

- 1 Идея метапрограммирования и Template Haskell
- 2 **Пример: генерация функций-проекторов для кортежей**
- 3 Реификация и квазицитирование



# Постановка задачи

```
fst :: (a, b) -> a
```

```
snd :: (a, b) -> b
```

- `proj 3 1 (x,y,z)` – должно возвращать `y`
- `proj 4 3 (x,y,z,w)` – должно возвращать `w`
- Каков тип `proj`?

```
proj :: Int -> Int -> ??? -> ???
```

- Эта информация известна на этапе компиляции, но Haskell не может ей воспользоваться.

## Пример: проекторы для 17-элементных кортежей

```

proj_17_0 (x0,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_) = x0
proj_17_1 (_,x1,_,_,_,_,_,_,_,_,_,_,_,_,_,_) = x1
proj_17_2 (_,_,x2,_,_,_,_,_,_,_,_,_,_,_,_,_) = x2
proj_17_3 (_,_,_,x3,_,_,_,_,_,_,_,_,_,_,_,_) = x3
proj_17_4 (_,_,_,_,x4,_,_,_,_,_,_,_,_,_,_,_) = x4
proj_17_5 (_,_,_,_,_,x5,_,_,_,_,_,_,_,_,_,_) = x5
proj_17_6 (_,_,_,_,_,_,x6,_,_,_,_,_,_,_,_,_) = x6
proj_17_7 (_,_,_,_,_,_,_,x7,_,_,_,_,_,_,_,_) = x7
proj_17_8 (_,_,_,_,_,_,_,_,x8,_,_,_,_,_,_,_) = x8
proj_17_9 (_,_,_,_,_,_,_,_,_,x9,_,_,_,_,_,_) = x9
proj_17_10 (_,_,_,_,_,_,_,_,_,_,x10,_,_,_,_,_) = x10
proj_17_11 (_,_,_,_,_,_,_,_,_,_,_,x11,_,_,_,_) = x11
proj_17_12 (_,_,_,_,_,_,_,_,_,_,_,_,x12,_,_,_) = x12
proj_17_13 (_,_,_,_,_,_,_,_,_,_,_,_,_,x13,_,_) = x13
proj_17_14 (_,_,_,_,_,_,_,_,_,_,_,_,_,_,x14,_) = x14
proj_17_15 (_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,x15,_) = x15
proj_17_16 (_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,x16) = x16

```

- Можно написать генератор функций-проекторов для кортежей любых размеров.

## Идея генератора функций-проекторов

```

{-# LANGUAGE TemplateHaskell #-}
module Pr where
import Language.Haskell.TH

proj :: Int -> Int -> Q Exp

```

- Функция-проектор `proj 17 10` — это функция, принимающая на вход кортеж из 17 элементов и возвращающая его компонент с индексом 10.

```

import Pr
main = do
  putStrLn $ $(proj 3 1) (undefined,"Success!",undefined)
  putStrLn $ $(proj 4 2) (undefined,undefined,"Success!",
                                                                    undefined)
  putStrLn $ $(proj 5 4) (undefined,undefined,undefined,
                                                                    undefined,"Success!")

```

## Как строить нужные функции?

```
ghci> runQ [| \ (x, _, _) -> x |]  
LamE [TupP [VarP x_0, WildP, WildP]] (VarE x_0)
```

- Анонимная функция (LamE), список её параметров и тело (возвращаемое значение)
- Кортеж образцов (pattern) и образцы для компонентов кортежа
- Имена переменных

### План решения

- 1 Вычисляем список образцов для компонентов кортежа нужного размера
- 2 Вычисляем тело
- 3 Строим анонимную функцию

## Пример построения для конкретного случая (с созданием имени)

```
proj 3 1 = do
  x <- newName "x"
  let args = [WildP,VarP x,WildP]
  return $ LamE [TupP args] (VarE x)
```

## Общий случай

```
proj n k = do
  x <- newName "x"
  let mkPat j
      | j == k = VarP x
      | otherwise = WildP
  return $ LamE [TupP $ map mkPat [0..n-1]] (VarE x)
```

## Версия с контролем корректности

```

proj n k = do
  | n > 1 && 0 <= k && k < n = do
    x <- newName "x"
    let mkPat j
          | j == k = VarP x
          | otherwise = WildP
        return $ LamE [TupP $ map mkPat [0..n-1]] (VarE x)
  | otherwise = fail "impossible projection"

```

- fail: ошибка будет показана во время компиляции.
- Проблема: мы строим AST вручную, можно попробовать использовать цитаты.

## Версия с цитатами

```

proj n k
| n > 1 && 0 <= k && k < n = do
  x <- newName "x"
  let mkPat j
      | j == k = varP x
      | otherwise = wildP
  [| \ $(tupP $ map mkPat [0..n-1]) -> $(varE x) |]
| otherwise = fail "impossible projection"

```

- Используются функции `varP`, `wildP`, `tupP`
- Нет конструктора `LamE` — вместо него анонимная функция внутри цитаты (пробел после `\` важен)
- Внедрение кода (`$(...)`) внутрь цитат
- Нет `return` — возвращается цитата (`[|...|]`)

## Генерация определений

```

projections :: Int -> Q [Dec]
projections n = fmap concat $ mapM mkDecl [0..n-1] where
  mkDecl k = do
    let nm = mName $ "proj_" ++ show n ++ "_" ++ show k
        [d| $(varP nm) = $(proj n k) |]

mkProjections :: [Int] -> Q [Dec]
mkProjections = fmap concat . mapM projections

```

- Функция mName
- Цитаты для определений (`[d|...|]`)



# Использование сгенерированных определений

```
{-# LANGUAGE TemplateHaskell #-}  
  
import Pr  
  
$(mkProjections [2..10])  
  
main = do  
  putStrLn $ proj_3_1 (undefined, "Success!", undefined)  
  putStrLn $ proj_4_2 (undefined, undefined, "Success!", undefined)  
  putStrLn $ proj_5_4 (undefined, undefined, undefined, undefined,  
                      "Success!")
```

```
ghci> :main  
Success!  
Success!  
Success!
```

## Вывод сгенерированного кода

```
$ ghc projector.hs -ddump-splices
[2 of 2] Compiling Main      ( projector.hs, projector.o )
projector.hs:7:3-23: Splicing declarations
  mkProjections [2 .. 10]
=====>
proj_2_0 = \ (x_a45F, _) -> x_a45F
proj_2_1 = \ (_, x_a45G) -> x_a45G
proj_3_0 = \ (x_a45H, _, _) -> x_a45H
proj_3_1 = \ (_, x_a45I, _) -> x_a45I
proj_3_2 = \ (_, _, x_a45J) -> x_a45J
proj_4_0 = \ (x_a45K, _, _, _) -> x_a45K
proj_4_1 = \ (_, x_a45L, _, _) -> x_a45L
proj_4_2 = \ (_, _, x_a45M, _) -> x_a45M
proj_4_3 = \ (_, _, _, x_a45N) -> x_a45N
...
```

# Содержание

- 1 Идея метапрограммирования и Template Haskell
- 2 Пример: генерация функций-проекторов для кортежей
- 3 Реификация и квазицитирование**

# Реификация

Реификация (интроспекция) — способность внедряемого кода исследовать структуру любых именованных сущностей.

- 1 Имя
- 2 Внутреннее устройство
- 3 Генерация кода

# Пример

```
data Complex = Polar Double Double  
             | Rectangular Double Double
```

- Полезно уметь определять, с помощью какого именно конструктора построено значение:

```
isPolar :: Complex -> Bool  
isRectangular :: Complex -> Bool
```

- В каждом конкретном случае написать такие функции (предикаты) несложно, но лень!
- Хочется написать генератор: изучаем строение заданного типа и для каждого конструктора генерируем определение предиката.

## Этап 1: реификация

```
{-# LANGUAGE TemplateHaskell #-}

module Predicates (mkPredicates) where

import Language.Haskell.TH

mkPredicates :: Name -> Q [Dec]
mkPredicates name = do
    TyConI (DataD _ _ _ _ constructorRecords _) <- reify name
    fmap concat $ mapM mkPredicate constructorRecords

reify :: Name -> Q Info
```

## Этап 2: генерация определений

```

mkPredicate :: Con -> Q [Dec]
mkPredicate (NormalC name types) =
  [d|
    $predicate = \z -> case z of
                        $pat -> True
                        _ -> False
  |]
where
  predicate = varP $ mkName $ "is" ++ nameBase name
  pat = conP name $ replicate (length types) wildP

```

- Внедрение внутри цитат

## Использование предикатов

```

{-# LANGUAGE TemplateHaskell #-}
module Main where
import Predicates

data Complex = Polar Double Double | Rectangular Double Double

data Shape = Circle Double | Square Double
           | Triangle Double Double Double

$(mkPredicates 'Complex) -- 'Complex - Name для типа Complex
$(mkPredicates 'Shape)

main = mapM_ print [isPolar c1, isRectangular c2,
                   isCircle s1, isSquare s2, isTriangle s3]
  where
    c1 = Polar 5 10; c2 = Rectangular 2 3
    s1 = Circle 4; s2 = Square 10; s3 = Triangle 1 1 1

```



# Виды цитат

- `[|...|]`, `[e|...|]` — выражения (expressions)
- `[d|...|]` — определения (declarations)
- `[t|...|]` — типы (types)
- `[p|...|]` — образцы (patterns)
  
- `[quoter|...|]` — квазицитата (quasiquotation)
- Для разбора содержимого квазицитат можно использовать произвольные парсеры.

## Пример: квазицитаты для многострочных литералов

```

module Str where

import Language.Haskell.TH
import Language.Haskell.TH.Quote

str = QuasiQuoter
    { quoteExp = stringE -- всё содержимое цитаты
                        -- будет одной строкой
    , quotePat = undefined
    , quoteType = undefined
    , quoteDec = undefined
    }

```

```

quoteExp :: String -> Q Exp
quotePat :: String -> Q Pat
quoteType :: String -> Q Type
quoteDec :: String -> Q [Dec]

```

```
{-# LANGUAGE QuasiQuotes #-}  
import Str  
  
verse :: String  
verse = [str|Дым табачный воздух выел.  
Комната -  
глава в крученыховском аде.  
Вспомни -  
за этим окном  
впервые  
руки твои, иступленный, гладил. |]
```

```
ghci> putStrLn verse  
Дым табачный воздух выел.  
Комната -  
глава в крученыховском аде.  
Вспомни -  
за этим окном  
впервые  
руки твои, иступленный, гладил.
```

# Template Haskell: заключение

- Выявление структуры именованных сущностей (reify)
- Явное построение абстрактных синтаксических деревьев (AST)
- Цитаты — преобразование кода на Haskell в AST
- Манипулирование AST
- Внедрение AST в код (splicing)
- Квазицитирование — произвольное преобразование содержимого цитаты

① GHC User's Guide

[https://downloads.haskell.org/~ghc/latest/docs/html/users\\_guide/](https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/)