Макросы вообще и в Haskell в частности

Михаил Беляев

13 декабря 2019 г.

Макросы в языках программирования

Инструменты генерации кода, встроенные в сам язык/компилятор

Какие вы знаете системы макросов и в каких языках?

CPP или препроцессор языка C

- Заимствовал идеи из макросистемы m4
- Имеет всем известные проблемы
- Зато универсален!

Забавный факт:

- Тьюринг-полный из-за багов реализации, хотя таковым не планировался
  - Т.е. на нём можно написать цикл, но выглядеть он будет ужасно и работать медленно

CPP в Haskell

Как ни странно, GHC поддерживает препроцессор в полной мере

```
{-# LANGUAGE CPP #-}
module Foo where

#define INT_SIZE 4

foo (INT_SIZE) = 2
foo _ = 4
```

CPP

Плюсы

- Простота на грани тупизны

Минусы

- Препроцессор вообще ничего не знает про язык
  - «Загрязнение» пространства имён
  - При подстановке может сломаться чужой код
  - Можно генерировать вообще некорректную чушь
- Сложные вещи выглядят сложно
- Если вы не в C/C++ или Haskell, лучше возьмите какой-нибудь шаблонизатор
  - StringTemplate
  - cog

Процедурные (или функциональные) макросы

- Есть некий набор типов данных для представления всего языка
  - По сути, это абстрактное синтаксическое дерево
- Макрос — это функция, которая принимает какие-то параметры и возвращает кусок дерева
- Как он подставляется в код — вопрос конкретного языка

Реализации

- Большинство диалектов LISP
- Scala
- Template Haskell

Процедурные (или функциональные) макросы

Плюсы

- Полный контроль над тем, что в итоге получается
- Максимальная мощность
- Получившееся будет гарантированно кодом на заданном языке

Минусы

- Адски непонятный код реализации, который не похож на то, что в итоге генерируется

Декларативные макросы (aka macro-by-example)

- Используя механизмы, похожие на pattern matching, задаём синтаксическую развертку кода
- При этом код макроса максимально близок к генерируемому

Реализации

- Scheme
- Rust
- Julia

Декларативные макросы: пример на Rust

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}

vec![1, 2, 3, 4]
```

Декларативные макросы (aka macro-by-example)

Плюсы

- Получившееся будет гарантированно кодом на заданном языке
- Больше возможностей, чем у препроцессора
- Код реализации близок к генерируемому и поэтому читается

Минусы

- Сам механизм macro-by-example вносит свои ограничения
- Как правило, не всё можно так сгенерировать

Экзотические вариации

Макросистема языка OCaml (в виде отдельного инструмента Camlp4/Camlp5) позволяет написать расширение языка путём модификации **грамматики** языка OCaml

Улучшения системы макросов

- Гигиенические макросы
  - Гарантируют, что генерируемые определения не мешают другим определениям
  - Не всегда удобно
- Квотирование/квазиквотирование
  - DSL для вписывания кода «как есть» в генерируемое дерево
- Реификация
  - Предоставление макросам информации от компилятора

Что из этого есть в перечисленных выше подходах?

Что позволяют генерировать макросы?

- Только выражения
- Только декларации
- Всё

Template Haskell

- Расширение языка Haskell
- Позволяет описывать функциональные макросы на Haskell
- Гигиена, квотирование, квазиквотирование, реификация
  - И много чего ещё
- Позволяют генерировать объявления, типы, паттерны, выражения

Template Haskell

```haskell
{-# LANGUAGE TemplateHaskell #-}

module Ex1 where
import Language.Haskell.TH
generate :: Int -> [Dec]
generate i = [FunD (mkName name) [Clause [] body []]]
                where
                    name = "fun" ++ show i
                    body = NormalB $ LitE $ StringL name
-- в другом файле:
$(generate 1) -- fun1 = "fun1"
$(generate 20) -- fun20 = "fun20"
```

Template Haskell: библиотека

- `Language.Haskell.TH`:
  - `Exp`, `Dec`, `Pat` ...
  - Содержит полное описание языка, вместе со всеми расширениями
  - Можно даже генерировать новые макросы!
  - За подробностями — в доки

Template Haskell: гигиена

- Макрос выше был **не гигиеничным**
  - Потому, что имена, которые он генерирует, видны снаружи

```
mkName -> newName
```

Template Haskell: гигиена

- Макрос выше был **не гигиеничным**
  - Потому, что имена, которые он генерирует, видны снаружи

```
mkName -> newName
```

```
:t newName
> newName :: String -> Q Name
```

Что ещё за Q?

Template Haskell: монада `Q`

- Чтобы генерировать «не занятые» имена, нужно какое-то состояние
- Оно (и не только оно) завёрнуто в отдельную монаду `Q` (Quoted)
- Она работает по-разному в зависимости от контекста
  - В интерпретаторе одно, в компиляторе другое
  - Подробности нас сейчас не очень интересуют

Template Haskell: монада Q

- Реализует всех наших старых друзей
  - `Monad`, `Functor`, `Applicative` и т.д.
- Для удобства все конструкторы продублированы функциями над `Q`

```
:t TupE
> TupE :: [Exp] -> Exp
:i tupE
> tupE :: [Q Exp] -> Q Exp
```

Template Haskell: гигиенический макрос

Получить элемент i из кортежа размера j

```
{-# LANGUAGE TemplateHaskell #-}
module Ex1 where
import Language.Haskell.TH
element i j =
    do
       let fname = mkName $ "element" ++ show i
       vnames <- mapM (\_ -> newName "x") [0..j - 1]
       let p = tupP (varP <$> vnames)
       let body = normalB $ varE (vnames !! i)
       (:[]) <$> funD fname [clause [p] body []]
```

Template Haskell: гигиенический макрос

Получить элемент i из кортежа размера j

```
$(element 2 3)
$(element 0 4)
```

```
>> ghci -XTemplateHaskell -ddump-splices Ex1a.hs
Ex1a.hs:6:3-13: Splicing declarations
    element 2 3
  ======>
    element2 (x_a4XL, x_a4XM, x_a4XN) = x_a4XN
Ex1a.hs:7:3-13: Splicing declarations
    element 0 4
  ======>
    element0 (x_a4Ym, x_a4Yn, x_a4Yo, x_a4Yp) = x_a4Ym
```

Template Haskell: квотирование

- Генерировать код в виде дерева сложно
- Он не похож на итоговый код
- С этим призвано бороться квотирование

Template Haskell: квотирование имён

- `mkName "x"` ⇒ `'x`
- `mkName "X"` ⇒ `''X`
- Только для известных имён
- Автоматически раскрывает имена до полных из пакета

```
> 'tail
GHC.List.tail
> ''Either
Data.Either.Either
```

Template Haskell: квотирование выражений

- Используется специальный синтаксис
  - `[e|x|]`, `[d|x|]`, `[t|x|]`, `[p|x|]`
- Результат это `Q Exp`/`[Q Dec]`/`Q Type`/`Q Pat`

Template Haskell: квотирование выражений

- Есть типы `Type0`/`Type1`/`Type2`...
- Есть функции их показа `showType$N`
- Нужно сгенерировать `instance Show`
- Но дерево получится гигантское!

Template Haskell: квотирование выражений

```
module Ex1 where
import Language.Haskell.TH
genShow :: Name -> Name -> Q [Dec]
genShow tp func =
    do
       let con = conT tp
       let body = varE func
       [d| instance () => Show $con where
                 show x = ($body) x |]
```

Template Haskell: квотирование выражений

Вызываем!

```
data Whatever = ...
showWhatever x = ...

$(genShow ''Whatever 'showWhatever)

Ex1a.hs:9:3-28: Splicing declarations
    genShow ''Type2 'showType2
  ======>
    instance Show Type2 where
      show x_a4oQ = showType2 x_a4oQ
```

Template Haskell: реификация

- Есть тип `Type`
- Мы хотим сгенерировать для него `Show`
  - Типа как это делает `deriving Show`
- Но у нас нет про него информации

```
> :t reify
reify :: Name -> Q Info
```

Template Haskell: реификация

```
> reify ''Maybe
TyConI (
    DataD []
          GHC.Base.Maybe
          [KindedTV a_3530822107858468865 StarT]
          Nothing
          [
            NormalC GHC.Base.Nothing [],
            NormalC GHC.Base.Just [
                (Bang NoSourceUnpackedness NoSourceStrictness,
                VarT a_3530822107858468865)
            ]
          ]
          []
)
```

Template Haskell: реификация

- Реификация позволяет автоматически генерировать инстансы для произвольных типов в compile time
- Надо помнить, что это — достаточно дорогая операция

Template Haskell: квазиквотирование

- Как квотирование, только в скобках может быть написано всё, что угодно

```
x = [whatever| (2 + x) * 15 $ # y |]
```

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE QuasiQuoting #-}

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

Template Haskell: квазиквотирование

```haskell
data QuasiQuoter
  = QuasiQuoter {
        quoteExp :: String -> Q Exp,
        quotePat :: String -> Q Pat,
        quoteType :: String -> Q Type,
        quoteDec :: String -> Q [Dec]
    }

whatever :: QuasiQuoter
whatever = ...
```

Template Haskell: квазиквотирование

- Попробуем реализовать на Haskell `printf`/`format`
- Пусть он в зависимости от числа плейсхолдеров в формате генерирует лямбда-выражение с нужным числом параметров

```
foo = [printf|{} + {}|] 4 5 -- "4 + 5"
```

Template Haskell: printf

```haskell
printf :: QuasiQuoter
printf = QuasiQuoter {
    quoteExp  = compilePrintf
  , quotePat  = notHandled "patterns"
  , quoteType = notHandled "types"
  , quoteDec  = notHandled "declarations"
} where notHandled e = error $ "Cannot use printf as " ++ e
```

Template Haskell: printf

```haskell
data PrintfElement = PureString String | EmbeddedValue Int deriving (Show, Eq)
type Printf = [PrintfElement]

parsePrintf :: Int -> String -> [PrintfElement]
parsePrintf i [] = []
parsePrintf i ('{':'}':rest) = EmbeddedValue i : parsePrintf (i + 1) rest
parsePrintf i (h:t) = let tl = parsePrintf i t in
                      case tl of
                           [] -> [PureString [h]]
                           (PureString s : ttl) ->
                                PureString (h:s) : ttl
                           (EmbeddedValue i : ttl) ->
                                PureString [h] : EmbeddedValue i : ttl
```

Template Haskell: printf

```haskell
compilePrintf :: String -> Q Exp
compilePrintf input = do
    let parsed = parsePrintf 0 input
    let isEmb (EmbeddedValue _) = True
        isEmb _ = False
    let argN = length $ filter isEmb parsed
    let xs = mkName <$> ("x" ++) <$> show <$> [0..argN - 1]
    let toExpr (PureString s) = stringE s
        toExpr (EmbeddedValue i) = appE (varE 'show) $ varE $ xs !! i
    let exprs = map toExpr parsed
    parensE $ lamE (varP <$> xs) (appE (varE 'concat) $ listE exprs)
```

Template Haskell: printf

```
> [printf| {} + {} |] 3 4
<interactive>:1:9-19: Splicing expression
    quoteExp printf " {} + {} "
  ======>
    (\ x0 x1 -> concat [" ", show x0, " + ", show x1, " "])
" 3 + 4 "

> [printf|{}--{}|] "Hello" 3.2
<interactive>:2:9-16: Splicing expression
    quoteExp printf "{}--{}"
  ======>
    (\ x0 x1 -> concat [show x0, "--", show x1])
"\"Hello\"--3.2"
```

Template Haskell: printf

- Разумеется, это очень простой пример
- Но когда на входе строка, можно развернуться
  - Библиотека Shakespeare генерирует сверхоптимизированный код из HTML, CSS, JS
  - Разные люди добавляют в Haskell другие фичи, например, интерполяцию
- Можно даже парсить Haskell внутри QQ с помощью `Language.Haskell.Ext`!
  - Yo dawg, I heard you liked haskell, so I put some haskell in your haskell...

Template Haskell: итог

- Процедурные макросы — это очень мощный механизм
- Непонятность можно снизить с помощью квотирования
- Позволяет реализовывать compile-time DSLs с помощью квазиквотирования
- Для типичных задач («сгенерируй мне экземпляр класса по заданному шаблону») всё-таки слишком много мороки
    - Есть библиотеки, которые позволяют от неё избавиться, например, `syb`

http://kspt.icc.spbstu.ru/course/lang
belyaev@kspt.icc.spbstu.ru

КСПТ | КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И ПРОГРАММНЫХ ТЕХНОЛОГИЙ СПбПУ

ПОЛИТЕХ
Санкт-Петербургский политехнический университет Петра Великого