Построение обобщённого зиппера средствами generics-sop

А. С. Болотина

Южный федеральный университет
Институт математики, механики и компьютерных наук им. И. И. Воровича
Кафедра информатики и вычислительного эксперимента

25 мая 2017

Семинар «Языки программирования и компиляторы»

Содержание

Обобщённое программирование типов данных

Алгебраические типы данных (АТД)

Алгебраический тип данных — это составной тип, который может быть представлен в виде типа-суммы из типов-произведений.

Тип-сумма

```
data StudentStage = Freshman
                  | Sophomore
                  | Junior
                  | Senior
```

Тип-произведение

```
data Person
     = Person String Int
              -- Name, Age
```

Тип арифметического выражения

```
data Expr = LInt Int
          | LBool Bool
          | Add Expr Expr
          | If Expr Expr Expr
```

Библиотека generics-sop

generics-sop — библиотека, поддерживающая обобщённое программирование типов данных в Haskell.

Особенности

- Каждый тип — N-арная сумма.
- Компоненты суммы — N-арные произведения.
- Использование современных расширений системы типов Haskell: *DataKinds*, *ConstraintKinds*, *PolyKinds*, *GADTs*, *RankNTypes*, ...
- Отделение метаданных от основного структурного представления типа.

N-арные произведения и суммы

Пример *N*-арного произведения: гетерогенный список

```
hlist :: NP I '[Bool, Int, Char]          -- тип
hlist = I False :* I 3 :* I 'x' :* Nil    -- терм
```

Пример *N*-арной суммы: выбор из списка

```
type HChoice = NS I '[Char, Bool, Int, Bool]   -- тип

c0, c2 :: HChoice
c0 = Z (I 'a')                                 -- термы
c2 = S (S (Z (I 13)))
```

Обобщённое представление АТД

Класс обобщённо представимых типов

```
type Rep a = NS (NP I) (Code a) -- N-арная сумма
                                -- произведений

class Generic (a :: *) where
    type Code a :: [[*]]
    from :: a -> Rep a
    to   :: Rep a -> a
```

Пример обобщённого представления

```
data Expr
    = LInt Int
    | LBool Bool
    | Add Expr Expr
    | If Expr Expr Expr
```

```
type RepExpr = NS (NP I) (
    '[ '[Int]
     , '[Bool]
     , '[Expr, Expr]
     , '[Expr, Expr, Expr]
     ])
```

Автоматическая генерация обобщённого представления

Полное определение класса Generic

```
class (All SListI (Code a)) => Generic (a :: *) where
    type Code a :: [[*]]
    type Code a = GCode a

    from          :: a -> Rep a
    default from :: (GFrom a, GHC.Generic a,
                     Rep a ~ NS (NP I) (GCode a))
                    => a -> Rep a
    from = gfrom

    to          :: Rep a -> a
    default to :: (GTo a, GHC.Generic a,
                   Rep a ~ NS (NP I) (GCode a))
                  => Rep a -> a
    to = gto
```

Пример

АТД: бинарное дерево

```
data BinTree a = Leaf a | Node (BinTree a) (BinTree a)
    deriving GHC.Generic

instance Generic (BinTree a)
```

Автоматически генерируемый код

```
instance Generic (BinTree a) where
    type Code (BinTree a) = '[ '[a],
                               '[BinTree a, BinTree a]
                             ]

    from :: BinTree a -> Rep (BinTree a)
    from (Leaf x)   = Z (I x :* Nil)
    from (Node l r) = S (Z (I l :* I r :* Nil))

    to :: Rep (BinTree a) -> BinTree a
    to (Z (I x :* Nil))             = Leaf x
    to (S (Z (I l :* I r :* Nil))) = Node l r
```

Структура данных «Зиппер»

Зиппер — структура данных, используемая для эффективной, чисто функциональной навигации по древовидной структуре.

Задача навигации

- **Задача:** представление древовидной структуры данных вместе с фокусом на текущем узле, который может перемещаться влево, вправо, вниз и вверх по этой структуре.
- **Решение:** фокус хранит текущий узел и путь, восходящий от него к корню дерева.

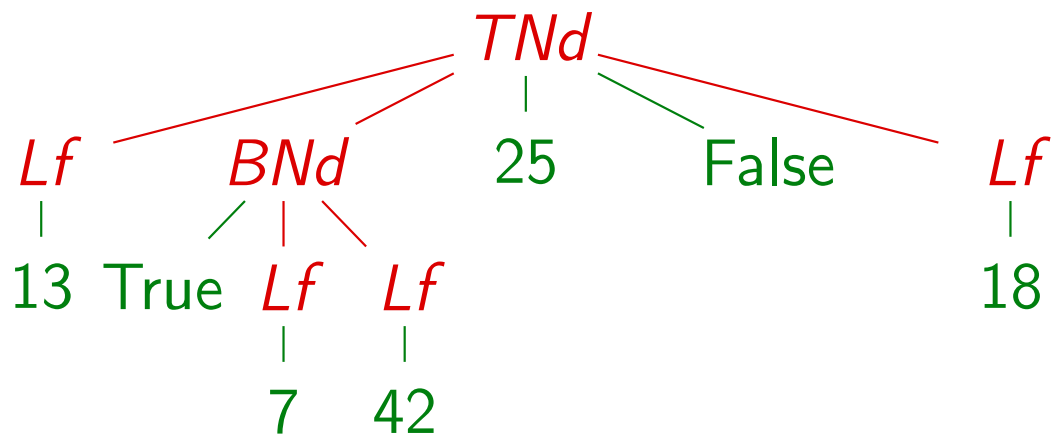
Мотивирующий пример (начало)

АТД: дерево

```
data Tree = Leaf  Int
          | TNode Tree Tree Int Bool Tree
          | BNode Bool Tree Tree
```

Пример дерева

```
tree :: Tree
tree
  = TNode (Leaf 13)
          (BNode True
                 (Leaf 7)
                 (Leaf 42))
          25
          False
          (Leaf 18)
```



Мотивирующий пример

Тип дерева Tree

```
data Tree = Leaf  Int
          | TNode Tree Tree Int Bool Tree
          | BNode Bool Tree Tree
```

Тип контекста для типа Tree

```
data TreeContext = TNode1 Tree Int Bool Tree
                 | TNode2 Tree Int Bool Tree
                 | TNode3 Tree Tree Int Bool
                 | BNode1 Bool Tree
                 | BNode2 Bool Tree
```

Путь — это список контекстов.

Тип зиппера

```
data TreeZipper = (Tree, [TreeContext])
```

Автоматизация получения контекста — дифференцирование

Работа К. Макбрайда (2001)

“The Derivative of a Regular Type is its Type of One-Hole Contexts.”

$$
\begin{aligned}
\partial_x x &\mapsto 1 \\
\partial_x C &\mapsto 0 \\
\partial_x (F + G) &\mapsto \partial_x F + \partial_x G \\
\partial_x (F \times G) &\mapsto F \times \partial_x G + \partial_x F \times G \\
\partial_x (F|_{y=G}) &\mapsto \partial_x F|_{y=G} + \partial_y F|_{y=G} \times \partial_x G
\end{aligned}
$$

Поставленная задача

Исследовать возможность применения средств библиотеки обобщённого программирования *generics-sop* в задаче построения зипперов для автоматизации получения типа контекста.

Введение алгебраических операций над типами

- *Сложение N-арных сумм произведений* — соответствует конкатенации списков из списков типов.
- *Умножение типа на N-арную сумму произведений* — добавление типа в начало каждого внутреннего списка суммы.
- *Умножение N-арного произведения на сумму произведений* — конкатенация списка типов с каждым внутренним списком суммы.

Введение алгебраических операций над типами (1)

Сложение N-арных сумм произведений

```
type family (.++) (xs :: [[*]]) (ys :: [[*]]) :: [[*]]

type instance (x ': xs) .++ ys = x ': (xs .++ ys)
type instance '[]       .++ ys = ys
```

Пример

```
type ExampleSum
    = '[ '[Int, Bool]] .++ '[ '[Char, Bool, Int], '[Int]]
 -- = '[ '[Int, Bool], '[Char, Bool, Int], '[Int]]
```

Введение алгебраических операций над типами (2)

Умножение типа на N-арную сумму произведений

```
type family (.*) (x :: *) (ys :: [[*]]) :: [[*]]

type instance x .* (ys ': yss) = (x ': ys) ': (x .* yss)
type instance x .* '[]         = '[]
```

Пример 1

```
type ExampleProd = Int .* '[ '[Int, Bool], '[Char]]
                -- = '[ '[Int, Int, Bool], '[Int, Char]]
```

Пример 2: умножение на единицу

```
type ProdUnit = Char .* '[ '[]] -- = '[ '[Char]]
```

Пример 3: умножение на ноль

```
type ProdZero = Bool .* '[]     -- = '[]
```

Введение алгебраических операций над типами (3)

Умножение N-арного произведения на сумму произведений

```
type family (.**) (x :: [*]) (ys :: [[*]]) :: [[*]]

(x ': xs) .** yss = x .* (xs .** yss)
'[]       .** yss = yss
```

Пример

```
type ExamplePProd
    = '[Int, Bool] .** '[ '[Bool], '[Bool, Char]]
 -- = '[ '[Int, Bool, Bool], '[Int, Bool, Bool, Char]]
```

Введение приоритета операций

```
infixr 6 .++
infixr 7 .*
infixr 7 .**
```

Получение типа контекста

Дифференцирование N-арного произведения

```
type family DiffProd (a :: *) (xs :: [*]) :: [[*]] where
    DiffProd a '[]  = '[]
    DiffProd a '[a] = '[ '[]]
    DiffProd a '[x] = '[]
    DiffProd a (x ': xs)
        = xs .** DiffProd a '[x] .++ x .* DiffProd a xs
```

Генерация типа контекста

```
type family ToContext (a :: *) (code :: [[*]]) :: [[*]]

type instance ToContext a (xs ': xss)
    = DiffProd a xs .++ ToContext a xss
type instance ToContext a '[] = '[]
```

Пример работы

Тип дерева

```
data Tree = Leaf  Int
          | TNode Tree Tree Int Bool Tree
          | BNode Bool Tree Tree
    deriving GHC.Generic

instance Generic Tree
```

Определение типа обобщённого представления контекста

```
type RepTreeContext = NS (NP I) (ToContext Tree (Code Tree))
```

Автоматически генерируемый код

```
type RepTreeContext = NS (NP I)
    ('[ '[Tree, Int, Bool, Tree]
      , '[Tree, Int, Bool, Tree]
      , '[Tree, Tree, Int, Bool]
      , '[Bool, Tree]
      , '[Bool, Tree] ])
```

Результаты

Разработан механизм, позволяющий автоматизировать процесс построения обобщённого представления контекста, основанный на структурном представлении типа в *generics-sop*:

```
type family ToContext (a :: *) (code :: [[*]]) :: [[*]]
```

Обобщённое представление контекста

```
type RepTreeContext = NS (NP I) (
    '[ '[Tree, Int, Bool, Tree]
     , '[Tree, Int, Bool, Tree]
     , '[Tree, Tree, Int, Bool]
     , '[Bool, Tree]
     , '[Bool, Tree]
     ])

type RepTreeContext'
    = NS (NP I) (ToContext Tree (Code Tree))
```

$\mathrm{RepTreeContext} \sim \mathrm{RepTreeContext}'$

Продолжение работы: функции навигации

Движение вниз для дерева

```
type TreeZipper = (Tree, [RepTreeContext])

goDown :: TreeZipper -> Maybe TreeZipper
goDown (Leaf _, _) = Nothing
goDown (t,      cs) = Just (toFirst t, toCtxFirst t : cs)
```

Реализация обобщённой функции **toFirst**

```
toFirst :: Generic a => a -> a
toFirst t = toFirstNS (Proxy :: Proxy a) (from t)

toFirstNS :: proxy a -> NS (NP I) xss -> a
toFirstNS p (S ns) = toFirstNS p ns
toFirstNS p (Z np) = toFirstNP p np

toFirstNP :: proxy a -> NP I xs -> a
toFirstNP p (I x :* xs) = ?         -- Нельзя вывести тип x
toFirstNP _ Nil         = error "impossible"
```

Ссылки

1. G. Huet. The Zipper. *JFP*, 1997.
2. C. McBride. The derivative of a regular type is its type of one-hole contexts, 2001.
3. A. Rodriguez, S. Holdermans, A. Löh, and J. Jeuring. Generic Programming with fixed points for mutually recursive datatypes. *ICFP*, 2009.
4. R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *SCP*, 2004.
5. E. de Vries and A. Löh. True Sums of Products. *WGP*, 2014.
6. Исходный код доступен в Git-репозитории: https://github.com/Maryann13/Zipper