

Функциональное программирование

Лекция 13. Рекурсивные типы

Денис Николаевич Москвин

СПбАУ РАН

14.05.2015

- 1 Алгебраические типы
- 2 Рекурсивные типы
- 3 Катаморфизмы
- 4 Анаморфизмы

- 1 Алгебраические типы
- 2 Рекурсивные типы
- 3 Катаморфизмы
- 4 Анаморфизмы

- Будем строить типы из единичного типа $()$, который обозначим 1 .
- Дизъюнктивную сумму будем обозначать $+$.
- Например, булев тип будет иметь вид $1+1$.
- Трёхэлементный тип будет иметь вид $1+1+1$.
- Подразумевается эквивалентность типов $1+(1+1)$ и $(1+1)+1$.
- Часто удобны обозначения $2 \equiv 1+1$ и $3 \equiv 1+1+1$ и т.д.
- Будем различать элементы, помечая их натуральными числами. Например, три элемента типа 3 — это 0_3 , 1_3 и 2_3 .

- Вводится операция умножения типов $*$. Умножение типов определяется их декартовым произведением.
- Например, тип $2*3$ — множество пар, в которых первый элемент булев (типа 2), а второй — из трёхэлементного типа 3.
- $2*3 \equiv 6$ ($\equiv 1+1+1+1+1+1$)?
- Иногда такой взгляд полезен, поскольку несложно определить изоморфизм: $(i_2, j_3) = (3i + j)_6$.

Возведение типа в степень типа

- Вводится операция возведения типа в степень \wedge . Тип Y^X — это функциональный тип $X \rightarrow Y$.
- $3^2 \equiv 9$. Действительно, у нас есть три константные функции, три «возрастающие» и три «убывающие» из булева типа в тройки.

$$f0 = 0_2 \mapsto 0_3, \quad 1_2 \mapsto 0_3$$

$$f1 = 0_2 \mapsto 1_3, \quad 1_2 \mapsto 1_3$$

$$f2 = 0_2 \mapsto 2_3, \quad 1_2 \mapsto 2_3$$

$$f3 = 0_2 \mapsto 0_3, \quad 1_2 \mapsto 1_3$$

$$f4 = 0_2 \mapsto 0_3, \quad 1_2 \mapsto 2_3$$

$$f5 = 0_2 \mapsto 1_3, \quad 1_2 \mapsto 2_3$$

...

- Вводится операция абстракции по переменной типа $\lambda X. T[X]$.
- Например, хаскелевский тип `data Maybe x = Nothing | Just x` записывается так $\lambda X. 1 + X$
- **Запишите на языке теории типов типы Хаскелла:**
 - `Either`
 - `(,,)`
 - `(a,Bool)`
 - `a -> Bool`
- Такую систему (с возможностью построения операторов над типами) называют $\lambda\omega$.

- 1 Алгебраические типы
- 2 Рекурсивные типы**
- 3 Катаморфизмы
- 4 Анаморфизмы

Рекурсивный тип списка

Список $L = \text{List } A$ значений типа A это либо пустой список, либо одоэлементый, либо двухэлементный и т.д.

$$L = 1 + A + A^2 + A^3 + \dots$$

Можно ли это записать компактно?

Рекурсивный тип списка

Список $L = \text{List } A$ значений типа A это либо пустой список, либо одоэлементый, либо двухэлементный и т.д.

$$L = 1 + A + A^2 + A^3 + \dots$$

Можно ли это записать компактно? Да, в виде рекурсивного уравнения:

$$L = 1 + A * (1 + A + A^2 + A^3 + \dots)$$

$$L = 1 + A * L$$

Но это и есть определение списка из Хаскелла!

```
data List a = Nil | Cons a (List a)
```

Как решить рекурсивное уравнение на типы

$$L = 1 + A * L$$

Если ввести для типов комбинатор неподвижной точки `FIX`, то метод решения стандартен

$$L = (\lambda X. 1 + A * X) L$$

$$L = \text{FIX } \lambda X. 1 + A * X$$

Для списка $L = \text{List } A$, удовлетворяющего уравнению $L = 1 + A * L$, мы нашли решение в виде неподвижной точки

$$L = \text{FIX } \lambda X. 1 + A * X$$

Для конструкции $\text{FIX } \lambda X. T[X]$ часто используют обозначение $\mu X. T[X]$, тогда оператор списка List может быть записан как

$$\text{List } A = \mu X. 1 + A * X$$

$$\text{List} = \lambda A. \mu X. 1 + A * X$$

Запишите в виде μ -типов следующие рекурсивные типы

(1) Натуральные числа

```
data Nat = Zero | Succ Nat
```

Запишите в виде μ -типов следующие рекурсивные типы

(1) Натуральные числа

```
data Nat = Zero | Succ Nat
```

$$\text{Nat} = \mu X. 1 + X$$

(2) Двоичные деревья

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Запишите в виде μ -типов следующие рекурсивные типы

(1) Натуральные числа

```
data Nat = Zero | Succ Nat
```

$$\text{Nat} = \mu X. 1 + X$$

(2) Двоичные деревья

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

$$\text{Tree} = \lambda A. \mu X. 1 + A * X^2$$

Реализация на Хаскелле

Хаскелл (даже 98) позволяет задать для оператор фиксированной точки типов.

```
data Mu f = In (f (Mu f))
```

```
> :k Mu
Mu :: (* -> *) -> *
> :t In
In :: f (Mu f) -> Mu f
```

Сравните Mu с fix

```
fix :: (a -> a) -> a
fix f = f (fix f)
```


Функтор, описывающий структуру типа: $N = \lambda X. 1 + X$

```
data N x = Z | S x

instance Functor N where
  fmap g Z     = Z
  fmap g (S x) = S (g x)
```

Рекурсивный тип вводим через неподвижную точку функтора на уровне типов

```
type Nat = Mu N
```

Пример для `data Nat = Z | S Nat` (2)

```
data N x = Z | S x
type Nat = Mu N
```

Нерекурсивный функтор, тип «нарастает»

```
Z      :: N x
S Z    :: N (N x)
S (S Z) :: N (N (N x))
```

Тип `Nat` (то есть `Mu N`) как его неподвижная точка

```
In Z      :: Mu N
In (S (In Z))      :: Mu N
In (S (In (S (In Z)))) :: Mu N
```

Пример для data List a = Nil | Cons a (List a)

Функтор, описывающий структуру типа:

$$L = \lambda A. \lambda X. 1 + A * X$$

```
data L a l = Nil | Cons a l

instance Functor (L a) where
  fmap g Nil          = Nil
  fmap g (Cons a l) = Cons a (g l)
```

Рекурсивный тип вводим через неподвижную точку функтора на уровне типов

```
type List a = Mu (L a)
```

Пример для data List a = Nil | Cons a (List a) (2)

```
data L a l = Nil | Cons a l
type List a = Mu (L a)
```

Нерекурсивный функтор, тип «нарастает»

```
Nil           :: L a l
Cons 'i' Nil  :: L Char (L a l)
Cons 'h' $ Cons 'i' Nil :: L Char (L Char (L a l))
```

Тип List Char (то есть Mu (L Char)) как его неподвижная точка

```
In Nil           :: Mu (L a)
In $ Cons 'i' $ In Nil           :: Mu (L Char)
In $ Cons 'h' $ In $ Cons 'i' $ In Nil :: Mu (L Char)
```

- 1 Алгебраические типы
- 2 Рекурсивные типы
- 3 Катаморфизмы**
- 4 Анаморфизмы

Пересоберём рекурсивную структуру

Рассмотрим преобразование рекурсивного типа в себя:

```
copy :: Functor f => Mu f -> Mu f
copy (In x) = In $ fmap copy x
```

Утверждение: это преобразование есть тождество.

На примере выражения `In (S (In (S (In Z))))` типа `Nat`:

```
copy      (In (S (In (S (In Z)))))) → def copy
In (fmap copy (S (In (S (In Z)))))) → def fmap
In (S (copy      (In (S (In Z)))))) → def copy
In (S (In (fmap copy (S (In Z)))))) → def fmap
In (S (In (S (copy      (In Z)))))) → def copy
In (S (In (S (In (fmap copy Z)))))) → def fmap
In (S (In (S (In Z))))))
```

Понятие катаморфизма

```
copy :: Functor f => Mu f -> Mu f
copy (In x) = In $ fmap copy x
```

Напишем обобщение `copy`, которое заменяет

`In :: f (Mu f) -> Mu f` на некоторую `phi :: f a -> a`.

Получим обобщение понятия свёртки, *катаморфизм* [МН95]

```
cata :: Functor f => (f a -> a) -> Mu f -> a
cata phi (In x) = phi $ fmap (cata phi) x
```

Для данных функтора f и типа a функция $phi :: f a \rightarrow a$ известна как *f -алгебра*.

Пример f-алгебры: N-алгебра

```
phiN :: N Int -> Int
phiN Z      = 0
phiN (S n) = succ n
```

Применяя cata к этой алгебре, получим преобразователь

```
natToInt :: Nat -> Int
natToInt = cata phiN
```

Сессия GHCi

```
> natToInt $ In (S (In (S (In Z))))
2
```


Пример (L a)-алгебры

```
phiL :: L a [a] -> [a]
phiL Nil          = []
phiL (Cons e es) = e : es
```

```
listify :: List a -> [a]
listify = cata phiL
```

Сессия GHCi

```
> listify $ In Nil
[]
> listify $ In $ Cons 'i' $ In Nil
"i"
> listify $ In $ Cons 'h' $ In $ Cons 'i' $ In Nil
"hi"
```

Ещё пара списочных алгебр

```
phiLen :: L a Int -> Int
phiLen Nil          = 0
phiLen (Cons _ es) = 1 + es

phiLSum :: Num a => L a a -> a
phiLSum Nil         = 0
phiLSum (Cons e es) = e + es
```

Сессия GHCi

```
> cata phiLen $ In $ Cons 'h' $ In $ Cons 'i' $ In Nil
2
> cata phiLSum $ In $ Cons 2 $ In $ Cons 3 $ In Nil
5
```

- 1 Алгебраические типы
- 2 Рекурсивные типы
- 3 Катаморфизмы
- 4 Анаморфизмы**

Пересоберём рекурсивную структуру иначе

Введём операцию, обратную $\text{In} :: f (\text{Mu } f) \rightarrow \text{Mu } f$

```
out :: Mu f -> f (Mu f)
out (In x) = x
```

Рассмотрим преобразование рекурсивного типа в себя:

```
copy' :: Functor f => Mu f -> Mu f
copy' x = In $ fmap copy' (out x)
```

Утверждение: это преобразование есть тождество.

На примере выражения `In (S (In (S (In Z))))` типа `Nat`:

```

copy'                (In (S (In (S (In Z)))))) → def copy'
In (fmap copy' (out (In (S (In (S (In Z))))))) → def out
In (fmap copy'      (S (In (S (In Z)))))) → def fmap
In (S (copy'        (In (S (In Z)))))) → def copy'
In (S (In (fmap copy' (out (In (S (In Z))))))) → def out
In (S (In (fmap copy'      (S (In Z)))))) → def fmap
In (S (In (S (copy'        (In Z)))))) → def copy'
In (S (In (S (In (fmap copy' (out (In Z))))))) → def out
In (S (In (S (In (fmap copy'      Z)))))) → def fmap
In (S (In (S (In Z))))

```

Понятие анаморфизма

```
copy' :: Functor f => Mu f -> Mu f
copy' x = In $ fmap copy' (out x)
```

Напишем обобщение `copy'`, которое заменяет

`out :: Mu f -> f (Mu f)` на некоторую `psi :: a -> f a`.

Получим нотацию *анаморфизма*:

```
ana :: Functor f => (a -> f a) -> a -> Mu f
ana psi x = In $ fmap (ana psi) (psi x)
```

Для данных функтора `f` и типа `a` функция `psi :: a -> f a` известна как ***f**-коалгебра*.

Пример f-коалгебры: N-коалгебра

```
psiN :: Int -> N Int  
psiN 0 = Z  
psiN n = S (n-1)
```

Применяя ана к этой коалгебре, получим преобразователь

```
intToNat :: Int -> Nat  
intToNat = ana psiN
```

Сессия GHCi

```
> intToNat 3  
In (S (In (S (In (S (In Z))))))
```



Erik Meijer and Graham Hutton.

Bananas in space: extending fold and unfold to exponential types.

In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 324–333, New York, NY, USA, 1995. ACM.