Функциональное программирование
Лекция 14. Рекурсивные типы

Денис Николаевич Москвин

СПбАУ РАН

12.12.2017

План лекции

1. Алгебраические типы
2. Рекурсивные типы
3. Катаморфизм
4. Анаморфизм и гилеморфизм

План лекции

Сумма типов

- 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.

Произведение типов

- Декартово произведение типов `X` и `Y` обозначим `X*Y`.
- Например, тип `2*3` — множество пар, в которых первый элемент булев (типа `2`), а второй — из типа `3`.
- `2*3` $\cong$ `6` ($\equiv$ `1+1+1+1+1+1`). В каком смысле?

Произведение типов

- Декартово произведение типов `X` и `Y` обозначим `X*Y`.
- Например, тип `2*3` — множество пар, в которых первый элемент булев (типа `2`), а второй — из типа `3`.
- `2*3` $\cong$ `6` ($\equiv$ `1+1+1+1+1+1`). В каком смысле?
- Несложно определить биекцию: $(\mathrm{i}_2, \mathrm{j}_3) \rightleftarrows (3\mathrm{i} + \mathrm{j})_6$.
- В общем случае два типа `a` и `b` ***изоморфны***, если существуют взаимно-обратные функции

```
from :: a -> b
to   :: b -> a
```

такие, что

```
to . from == id      -- :: a -> a
from . to == id      -- :: b -> b
```

Экспоненциальные типы

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

$$
\begin{aligned}
\mathrm{f0} = &\quad 0_2 \mapsto 0_3, \quad 1_2 \mapsto 0_3 \\
\mathrm{f1} = &\quad 0_2 \mapsto 1_3, \quad 1_2 \mapsto 1_3 \\
\mathrm{f2} = &\quad 0_2 \mapsto 2_3, \quad 1_2 \mapsto 2_3 \\
\\
\mathrm{f3} = &\quad 0_2 \mapsto 0_3, \quad 1_2 \mapsto 1_3 \\
\mathrm{f4} = &\quad 0_2 \mapsto 0_3, \quad 1_2 \mapsto 2_3 \\
\mathrm{f5} = &\quad 0_2 \mapsto 1_3, \quad 1_2 \mapsto 2_3 \\
&\quad \ldots
\end{aligned}
$$

Все стандартные алгебраические свойства верны:
`Z^(X+Y)`$\cong$`Z^X*Z^Y`, `Z*(X+Y)`$\cong$`Z*X+Z*Y` и т.п.

Параметризованные типы

- Вводится переменные типа и операция абстракции по таким переменным: λ `X. T[X]`.
- Например, для типа Haskell

  `data Maybe x = Nothing | Just x`

  конструктор типа `Maybe` записывается так

  λ `X. 1 + X`
- Запишите на языке теории типов типы Haskell:
  - `Either`
  - `(,,)`
  - `(a,Bool)`
  - `(,) Bool`
  - `a -> Bool`
  - `(->) Bool`
- Такую систему (с возможностью построения операторов над типами) называют $\lambda\underline{\omega}$.

План лекции

Рекурсивный тип списка

Список `L = List A` значений типа `A` это либо пустой список, либо одоэлементый, либо двухэлементный и т.д.

```
L = 1 + A + A^2 + A^3 + ...
```

Можно ли это записать компактно?

Рекурсивный тип списка

Список `L = List A` значений типа `A` это либо пустой список, либо одоэлементый, либо двухэлементный и т.д.

```
L = 1 + A + A^2 + A^3 + ...
```

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

```
L = 1 + A * (1 + A + A^2 + A^3 + ...)
L = 1 + A * L
```

Но это и есть определение списка из Хаскелла!

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

Рекурсивные типы

Как решить рекурсивное уравнение на типы

```
L = 1 + A * L
```

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

```
L = (λX. 1 + A * X) L

L = FIX λX. 1 + A * X
```

μ-нотация

Для списка `L = List A`, удовлетворяющего уравнению `L = 1 + A * L`, мы нашли решение в виде неподвижной точки

```
L = FIX λX. 1 + A * X
```

Для конструкции `FIX λX. T[X]` часто используют обозначение `μX. T[X]`, тогда оператор списка `List` может быть записан как

```
List A = μX. 1 + A * X

List = λA. μX. 1 + A * X
```

Упражнения

Запишите в виде μ-типов следующие рекурсивные типы

(1) Натуральные числа

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

Упражнения

Запишите в виде μ-типов следующие рекурсивные типы

(1) Натуральные числа

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

```
Nat = μX. 1 + X
```

(2) Двоичные деревья

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

Упражнения

Запишите в виде μ-типов следующие рекурсивные типы

(1) Натуральные числа

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

```
Nat = μX. 1 + X
```

(2) Двоичные деревья

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

```
Tree = λA. μX. 1 + A * X^2
```

Реализация на Хаскелле

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

```
newtype Fix f = In (f (Fix f))
```

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

Сравните `Fix` с `fix`

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

Пример для `data Nat = Z | S Nat`

Функтор, описывающий структуру типа: `N = λX. 1 + X`

```
data N x = Z | S x

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

Тип `N` нерекурсивен.
Рекурсивный тип вводим через неподвижную точку функтора на уровне типов

```
type Nat = Fix N
```

Пример для `data Nat = Z | S Nat` (2)

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

Нерекурсивный функтор, тип «нарастает»

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

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

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

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

Функтор, описывающий структуру типа:
`L = λA. λX. 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 = Fix (L a)
```

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

```
data L a l = Nil | Cons a l
type List a = Fix (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 `Fix (L Char)`) как его неподвижная точка

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

План лекции

Пересоберём рекурсивную структуру

Рассмотрим преобразование рекурсивного типа в себя:

```
copy :: Functor f => Fix f -> Fix 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 (κατα — вниз)

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

Напишем обобщение `copy`, которое заменяет упаковку в `In :: f (Fix f) -> Fix f` на некоторую `phi :: f a -> a`.
Получим обобщение понятия свёртки, ***катаморфизм*** [MH95]

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

Для данных функтора `f` и типа `a` функция `phi :: f a -> a` известна как ***f-алгебра***. Тип `a` называют ***носителем*** (carrier).

```
type Algebra f a = f a -> a
cata :: Functor f => Algebra f a -> Fix f -> a
```

Пример f-алгебры: N-алгебра

```
phiN :: N Int -> Int    -- Algebra N 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

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

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

Сессия GHCi

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

Инициальная алгебра

Конструктор `In :: f (Fix f) -> Fix f` сам является алгеброй (с носителем `Fix f`)

```
phiIn = Algebra f (Fix f)
phiIn = In
```

Эта алгебра называется ***инициальной алгеброй***.
Инициальная алгебра сохраняет всю информацию о структуре, поданной на вход. Ее катаморфизм — тождественная функция

```
copy :: Functor f => Fix f -> Fix f
copy = cata phiIn
```

План лекции

Пересоберём рекурсивную структуру иначе

Введём операцию, обратную `In :: f (Fix f) -> Fix f`

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

Пара из `In` и `out` задает изоморфизм между типами `Fix f` и `f (Fix f)` (***f-изоморфизм***).
Рассмотрим преобразование рекурсивного типа в себя:

```
copy' :: Functor f => Fix f -> Fix f
copy' x = In $ fmap copy' $ out x
```

Утверждение: преобразование `copy'` 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 (αναα — вверх)

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

Напишем обобщение `copy'`, которое заменяет `out :: Fix f -> f (Fix f)` на некоторую `psi :: a -> f a`.
Получим ***анаморфизм***:

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

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

```
type Coalgebra f a = a -> f a
ana :: Functor f => Coalgebra f a -> a -> Fix f
```

Пример f-коалгебры: N-коалгебра

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

Применяя `ana` к этой коалгебре, получим преобразователь

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

Сессия GHCi

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

Терминальная коалгебра

Функция `out :: Fix f -> f (Fix f)` является коалгеброй (с носителем `Fix f`)

```
psiOut :: Coalgebra f (Fix f)
psiOut = out
```

Эта коалгебра называется ***терминальной коалгеброй***.
Ее катаморфизм — тождественная функция

```
copy' :: Functor f => Fix f -> Fix f
copy' = ana psiOut  -- == id
```

Понятие гилеморфизма (υλη — вещество, материя)

Гилеморфизм (hylomorphism) — последовательное применение анаморфизма, а затем катаморфизма:

```
hylo :: Functor f => Algebra f a -> Coalgebra f b -> b -> a
hylo phi psi = cata phi . ana psi
```

```
phiLProd ::  Algebra (L Integer) Integer
phiLProd Nil          = 1
phiLProd (Cons e es) = e * es

psiLToZero :: Coalgebra (L Integer) Integer
psiLToZero 0 = Nil
psiLToZero n = Cons n (n-1)

factorial :: Integer -> Integer
factorial = hylo phiLProd psiLToZero
```

Ката- и анаморфизмы суть гилеморфизмы

```
hylo :: Functor f => Algebra f a -> Coalgebra f b -> b -> a
hylo phi psi = cata phi . ana psi
```

Ката- и аноморфизмы легко выразить через гилеморфизм:

```
cata' :: Functor f => Algebra f a -> Fix f -> a
cata' phi = hylo phi out

ana' :: Functor f => Coalgebra f a -> a -> Fix f
ana' psi = hylo In psi
```

Литература

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.