Функциональное программирование
Лекция 14. Рекурсивные типы

Денис Николаевич Москвин

СПбГУ, факультет МКН,
бакалавриат «Современное программирование», 2 курс

11.12.2025

План лекции

План лекции

1. Структурное представление алгебраических типов
2. Структурное представление рекурсивных типов
3. Катаморфизм
4. Анаморфизм и гилеморфизм

Два подхода к дизайну формальных языков

- *номинальный*: построение и различение языковых конструкций происходит через пользовательские имена

```
data Point = Pt {ptX :: Double, ptY :: Double}
data AltII = Small Int | Large Integer
```

- *структурный*: пользователь ограничен в именовании; языковые конструкции строют с помощью предзаданных синтаксических элементов и различают по их структуре

```
(Double,Double)
Int | Integer -- impossible in Haskell
```

Сумма типов: стуктурный подход

- 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 $(1 + 1) + 1$ (чуть позже проясним).
- 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: стуктурный подход

- 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: стуктурный подход

- 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: $(i_2, j_3) \rightleftarrows (3i + j)_6$.
- В общем случае два типа a и b ***изоморфны***, если существуют взаимно-обратные функции

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

такие, что

```haskell
    to . from  ≡  id      -- :: a -> a
    from . to  ≡  id      -- :: b -> b
```

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

$$
\begin{array}{llll}
f0 = & 0_2 \mapsto 0_3, & 1_2 \mapsto 0_3 & \\
f1 = & 0_2 \mapsto 1_3, & 1_2 \mapsto 1_3 & \\
f2 = & 0_2 \mapsto 2_3, & 1_2 \mapsto 2_3 & \quad f6 = \; 0_2 \mapsto 1_3, \quad 1_2 \mapsto 0_3 \\
 & & & \quad f7 = \; 0_2 \mapsto 2_3, \quad 1_2 \mapsto 0_3 \\
f3 = & 0_2 \mapsto 0_3, & 1_2 \mapsto 1_3 & \quad f8 = \; 0_2 \mapsto 2_3, \quad 1_2 \mapsto 1_3 \\
f4 = & 0_2 \mapsto 0_3, & 1_2 \mapsto 2_3 & \\
f5 = & 0_2 \mapsto 1_3, & 1_2 \mapsto 2_3 &
\end{array}
$$

- 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: стуктурный подход

- Можно ввести в язык переменные типа и операцию абстракции по таким переменным: $\lambda\alpha.\, \mathrm{T}[\alpha]$.
- Например, для типа Haskell
  `data Maybe a = Nothing | Just a`
  конструктор типа `Maybe` записывается так
  $\lambda\alpha.\, 1 + \alpha$
- Запишите на языке структурных типов типы Haskell:
  - `Either`
  - `(,,)`
  - `(a,Bool)`
  - `(,) Bool`
  - `a -> Bool`
  - `(->) Bool`
- Систему типов с возможностью построения лямбда-абстракций над типами называют $\lambda\underline{\omega}$.

План лекции

Рекурсивный тип списка

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

$$L = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

Можно ли это записать компактно?

Рекурсивный тип списка

Список $L = \texttt{List}\ \alpha$ значений типа α это либо пустой список, либо одоэлементный, либо двухэлементный и т.д.

$$L = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

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

$$L = 1 + \alpha * (1 + \alpha + \alpha^2 + \alpha^3 + \ldots)$$

$$L = 1 + \alpha * L$$

Но это и есть определение списка из Haskell

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

Как решить рекурсивное уравнение на типы

$$L = 1 + \alpha * L$$

Если ввести для типов комбинатор неподвижной точки `FIX`, то можно воспользоваться стандартным методом

$$L = (\lambda\gamma.\, 1 + \alpha * \gamma)\; L$$

$$L = \texttt{FIX}\; \lambda\gamma.\, 1 + \alpha * \gamma$$

μ-нотация

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

$$\mathrm{L} = \mathtt{FIX}\ \lambda\gamma.\, 1 + \alpha * \gamma$$

Для конструкции $\mathtt{FIX}\ \lambda\gamma.\, \mathrm{T}[\gamma]$ часто используют обозначение $\mu\gamma.\, \mathrm{T}[\gamma]$, тогда тип списка $\mathtt{List}$ может быть записан как

$$\mathtt{List}\ \alpha = \mu\gamma.\, 1 + \alpha * \gamma$$

$$\mathtt{List} = \lambda\alpha.\, \mu\gamma.\, 1 + \alpha * \gamma$$

Примеры

Переведем в μ-нотацию натуральные числа

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

Примеры

Переведем в μ-нотацию натуральные числа

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

$$\mathtt{Nat} = 1 + \mathtt{Nat}$$
$$\mathtt{Nat} = \mu\gamma.\, 1 + \gamma$$

Переведем в μ-нотацию двоичные деревья

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

Примеры

Переведем в μ-нотацию натуральные числа

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

$$\mathtt{Nat} = 1 + \mathtt{Nat}$$
$$\mathtt{Nat} = \mu\gamma.\, 1 + \gamma$$

Переведем в μ-нотацию двоичные деревья

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

$$\mathtt{Tree}\ \alpha = 1 + \alpha * \mathtt{Tree}\ \alpha * \mathtt{Tree}\ \alpha$$
$$\mathtt{Tree}\ \alpha = \mu\gamma.\, 1 + \alpha * \gamma * \gamma$$
$$\mathtt{Tree} = \lambda\alpha.\, \mu\gamma.\, 1 + \alpha * \gamma^2$$

Основное характеристическое свойство неподвижной точки

$$\texttt{FIX}\ F = F\ (\texttt{FIX}\ F)$$

позволяет раскрывать μ-нотацию. Заменяем F на $\lambda\gamma.\,T$

$$\mu\gamma.\,T = (\lambda\gamma.\,T)\ \mu\gamma.\,T$$
$$\mu\gamma.\,T = [\gamma := \mu\gamma.\,T]\,T$$

- Если рассматривать обе части равенства как эквивалентные по определению, то говорят об ***эквирекурсивных типах***.
- Если рассматривать обе части равенства как изоморфные, задавая преобразующие функции, то говорят об ***изорекурсивных типах***.

Эквирекурсивные типы: структурно и номинально

Рекурсивные типы можно строить на основе отношения эквивалентности (***эквирекурсивные типы***)

$$\mu\gamma.\, T \;=\; [\gamma := \mu\gamma.\, T]\, T$$

Их удобно представлять как бесконечные деревья. Например, тип функции неограниченной арности

$$\mathtt{Hungry} \;\equiv\; \mu\gamma.\,\gamma^{\alpha} \;\equiv\; \mu\gamma.\,\alpha \to \gamma \;=\; \mu\gamma.\,\alpha \to (\mu\gamma.\,\alpha \to \gamma)$$

В явном (экви)рекурсивном виде $\mathtt{Hungry} = \alpha \to \mathtt{Hungry}$. Обитатель этого типа конструируется как неподвижная точка $\mathbf{K}$

$$\mathtt{eater} = \mathtt{fix}\ (\lambda f^{\mathtt{Hungry}}\, x^{\alpha}.\, f)$$

Проверьте, что `eater` имеет тип `Hungry`.

Рассмотрим тип $\mathtt{AutoA} = \mu\gamma.\,\alpha^{\gamma} \equiv \mu\gamma.\,\gamma \to \alpha$. Раскрывая μ-нотацию

$$\mu\gamma.\,\alpha^{\gamma} \;=\; \alpha^{\mu\gamma.\,\alpha^{\gamma}} \;\equiv\; (\mu\gamma.\,\alpha^{\gamma}) \to \alpha$$

То есть в явном (экви)рекурсивном виде $\mathtt{AutoA} = \mathtt{AutoA} \to \alpha$. Теперь можем, например, типизировать самоприменение и ω

$$\dfrac{\dfrac{\dfrac{x^{\mathtt{AutoA}} \vdash x : \mathtt{AutoA}}{x^{\mathtt{AutoA}} \vdash x : \mathtt{AutoA} \to \alpha}\,(\mathtt{AutoA} = \mathtt{AutoA} \to \alpha) \qquad x^{\mathtt{AutoA}} \vdash x : \mathtt{AutoA}}{x^{\mathtt{AutoA}} \vdash x\,x : \alpha}\,(\to E)}{\dfrac{\vdash \lambda x.\,x\,x : \mathtt{AutoA} \to \alpha}{\vdash \lambda x.\,x\,x : \mathtt{AutoA}}\,(\mathtt{AutoA} = \mathtt{AutoA} \to \alpha)}\,(\to I)$$

Попробуйте, используя $\mathtt{AutoA}$, приписать типы Ω и Y.

Изорекурсивные типы

Другой подход — ***изорекурсивные типы***, базирующиеся не на декларации эквивалентности, а на изоморфизме

$$\mu\gamma.\,\mathsf{T} \;\cong\; [\gamma := \mu\gamma.\,\mathsf{T}]\,\mathsf{T}$$

Изоморфизм задается парой функций

$$\begin{array}{lcl} \mathtt{out} & : & \mu\gamma.\,\mathsf{T} \to [\gamma := \mu\gamma.\mathsf{T}]\,\mathsf{T} \\ \mathtt{in} & : & ([\gamma := \mu\gamma.\,\mathsf{T}]\,\mathsf{T}) \to \mu\gamma.\,\mathsf{T} \end{array}$$

Вспоминая, что $\mu\gamma.\,\mathsf{T} \equiv \mathtt{FIX}\;\lambda\gamma.\,\mathsf{T}$, и вводя $\mathsf{F} = \lambda\gamma.\,\mathsf{T}$, имеем

$$\begin{array}{lcl} \mathtt{out} & : & \mathtt{FIX}\,\mathsf{F} \to \mathsf{F}\,(\mathtt{FIX}\,\mathsf{F}) \\ \mathtt{in} & : & \mathsf{F}\,(\mathtt{FIX}\,\mathsf{F}) \to \mathtt{FIX}\,\mathsf{F} \end{array}$$

Реализация изорекурсивных типов на Хаскелле

Haskell позволяет задать оператор `Fix` для типов

```
newtype Fix f = In { out :: f (Fix f) }
```

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

Пара из `In` и `out` задает изоморфизм между типами `Fix f` и `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: $\mathrm{N} = \lambda\gamma.\, 1 + \gamma$

```
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 Nat вводим не через прямую рекурсию, а 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:

```
Z                         :: N (Fix N) -- в частности
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: $L = \lambda\alpha.\,\lambda\gamma.\,1 + \alpha * \gamma$

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

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` (т.е. `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

1. Структурное представление алгебраических типов
2. Структурное представление рекурсивных типов
3. **Катаморфизм**
4. Анаморфизм и гилеморфизм

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

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

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

Применяя cata к этой алгебре, получим преобразователь

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

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

Пример (L a)-алгебры

```haskell
data L a x = Nil | Cons a x
type List a = Fix (L a)
```

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

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

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

Ещё пара списочных алгебр

```haskell
data L a x = Nil | Cons a x
type List a = Fix (L a)
```

```haskell
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
```

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

Обычный foldr через f-алгебры

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr fun ini []     = ini
foldr fun ini (x:xs) = fun x (foldr fun ini xs)
```

может быть приведен к (L a)-алгебраическому виду

```
foldr_ :: Algebra (L a) b -> [a] -> b
foldr_ phi [] = ini
  where ini = phi Nil
foldr_ phi (x:xs) = fun x (foldr_ phi xs)
  where fun a b = phi (Cons a b)
```

```
GHCi> foldr_ phiLLen "Hello"
5
GHCi> foldr_ phiLSum [1..3]
6
```

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.

copy' — это id

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 (ανα — 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

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

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

Применяя ana к этой коалгебре, получим преобразователь

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

```
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`)

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

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

```haskell
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

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

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

Ката- и анаморфизмы суть гилеморфизмы

```haskell
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:

```haskell
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.