

Введение в функциональное программирование
– Лекция 5. Алгебраические типы данных.
Сопоставление с образцом

Олег Смирнов
oleg.smirnov@gmail.com

14 октября 2011 г.

Содержание

| | | |
|---|--------------------------------------|----|
| 1 | Алгебраические типы данных | 2 |
| 2 | Сопоставление с образцом | 3 |
| 3 | Абстракция данных | 5 |
| 4 | Логика высказываний | 6 |
| 5 | Деревья и полиморфные типы данных | 7 |
| 6 | Списки как алгебраический тип данных | 9 |
| 7 | Тип необязательных значений | 10 |

Цель лекции

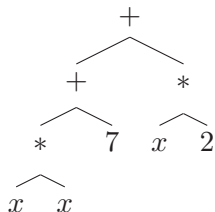
- Алгебраические типы данных и сопоставление с образцом
- Полиморфные типы данных
- Списки, деревья, тип *option*

1 Алгебраические типы данных

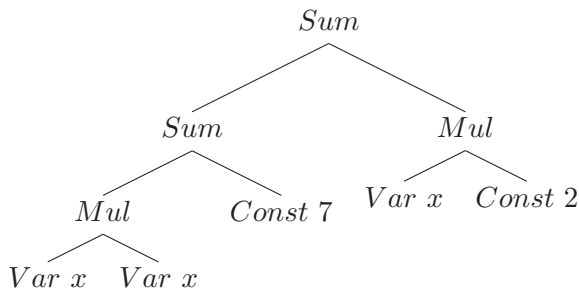
Пусть необходимо написать программу, которая бы оперировала арифметическими выражениями вида

$$2 * x + x * x + 7 \tag{1}$$

Такие выражения удобно представлять в виде синтаксических деревьев, где операции задаются узлами, их операнды – потомками, а константы хранятся в листьях дерева:



В наших выражениях листья могут быть двух типов: константы, заданные целым числом, и переменные, заданные строкой-именем переменной. Определим новый тип данных, удобный для представления арифметических выражений. Пусть переменные задаются с префиксом *Var*, константы с префиксом *Const*, а сумма и умножение – с префиксами *Sum* и *Mul* соответственно:



```
type Expr =  
  | Var of string  
  | Const of int  
  | Sum of Expr * Expr  
  | Mul of Expr * Expr
```

В такой нотации вертикальная черта обозначает выбор из нескольких вариантов, а знак умножения в строчках с *Sum* и *Mul* – просто пару из двух значений типа *Expr*. После такого определения можно записать выражение типа *Expr*:

```
let constSeven = Const(5)
let sumOneTwo = Sum(Const(1), Const(2))
```

Выражение 1 можно определить как:

```
Sum(Sum(Mul(Var("x"), Var("x")), Const(7)),
    Mul(Var("x"), Const(2)))
```

2 Сопоставление с образцом

Тип данных *Expr* определён как набор вариантов. Для работы с такой структурой удобно использовать оператор *match*, который сопоставляет аргумент с различными образцами. Напишем функцию, которая будет вычислять значение выражения:

```
let rec eval Expr =
  match Expr with
  | Const(c) -> c
  | Var(v) -> getVar v
  | Sum(e1, e2) -> eval e1 + eval e2
  | Mul(e1, e2) -> eval e1 * eval e2
```

В левой части каждой строки после оператора *match* находится *образец*. В правой части после стрелочки – действие, которое выполнится, если аргумент *match* совпадёт с данным образцом. В образцах можно использовать связывание переменных. Например, *e1* и *e2* будут связаны с аргументами *Mul* и *Sum*.

Расширим тип данных *Expr*, чтобы он позволял задавать не только сложение и умножение, но и другие бинарные операции. Каждая операция будет задана тремя аргументами: сигнатурой, первым аргументом и вторым аргументом. Аналогично определим унарные операции:

```
...
| Binary of Op2 * Expr * Expr
| Unary of Op1 * Expr
```

Здесь *Op2* и *Op1* – сигнатуры бинарных и унарных операций соответственно. Их можно определить в качестве самостоятельных типов данных:

```
Op2 = Sum | Sub | Mul | Div
Op1 = Neg
```

Теперь можно расширить функцию *eval* для поддержки новых операций:

```
let rec eval Expr =
  match Expr with
  ...
  | Binary(op, e1, e2) ->
      match op with
      | Sum -> eval e1 + eval e2
      | Mul -> eval e1 * eval e2
      | Div -> eval e1 / eval e2
      | Sub -> eval e1 - eval e2
  | Unary(op, e1) ->
      match op with
      | Neg -> not eval e1
      | _ -> eval e1
```

Допустим, на момент реализации известна только операция отрицания *Neg*. Последняя строчка функции – это “заглушка”, которая сработает для любого неизвестного значения *op*. Здесь нижнее подчёркивание означает “что угодно” – универсальный образец, подходящий для всех значений. Важно помнить, что правила *match* выполняются *сверху вниз*. Т.е. все правила, записанные после сопоставления с *_*, никогда не выполнятся.

Предположим, что функция *eval* должна работать только с положительными константами, а все отрицательные – “сбрасывать” в ноль. Это можно записать следующим правилом:

```
match Expr with
| Const(c) when c < 0 -> 0
| Const(c) -> c
```

Условие после ключевого слова *when* называется “стражником” от англ. guard.

Правила сопоставления с образцом можно комбинировать, делая код более читабельным. Функцию вычисления факториала можно записать так:

```
let rec fact n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | n -> n * fact (n - 1)
```

А можно короче, объединив в два условия с одинаковым действием:

```
let rec fact n =
  match n with
  | 0 | 1 -> 1
  | n -> n * fact (n - 1)
```

3 Абстракция данных

В такой нотации, как в примерах выше, *Const*, *Var* и т.п. называют *конструкторами*, т.к. они позволяют создать экземпляр данного типа. Кроме конструкторов ещё определяют *селекторы* и *предикаты*. Например, для типа *Expr* удобно написать предикат $isConst : Expr \rightarrow bool$ и $isVar : Expr \rightarrow bool$:

```
let isConst e =
  match e with
  | Const(_) -> true
  | _ -> false
```

```
let isVar e =
  match e with
  | Var(_) -> true
  | _ -> false
```

Примером селектора может быть функция *getConst*, которая возвращает значение, если её аргумент – константа.

Для работы с графами можно задать тип данных *Graph*, у которого конструкторами будут *Node* и *Link* – вершина и ребро графа, соответственно. А предикатом может быть функция *isConnected*, проверяющая, является ли граф связным.

Тип данных, заданный набором своих конструкторов, селекторов и предикатов, называют *абстрактным типом данных*, т.к. его внутреннее представление скрыто от программиста. Граф типа *Graph* может храниться в виде матрицы смежности, матрицы инцидентности или в виде списка вершин. Его *интерфейс* из конструкторов, селекторов и предикатов для любой реализации останется неизменным.

Тип данных *Expr* называется *алгебраическим типом данных*, т.к. он представлен в виде размеченного объединения декартовых произведений множеств или, другими словами, в виде *суммы произведений типов*. Действительно, вертикальная черта, объединяющая варианты выбора между *Var*, *Const* и др., эквивалентна логической операции “или” или суммированию. Знак умножения в аргументах конструкторов означает

декартово произведение и задаёт кортеж из соответствующих типов. А *Const*, *Var* и т.п. – это *метки* для таких кортежей.

4 Логика высказываний

Рассмотрим ещё один пример. Выражения *логики высказываний* можно определить с помощью константы *T* (“истина”), и операторов отрицания, конъюнкции и дизъюнкции:

```
type proposition =  
  | True  
  | Not of proposition  
  | And of proposition * proposition  
  | Or of proposition * proposition
```

Несколько примеров выражений, записанных с помощью такого типа данных:

1. $\neg T \vee \neg\neg T$

```
let prop1 = Or(Not(True), Not(Not(True)))
```

2. $\neg\neg\neg\neg T$

```
let prop2 = Not(Not(Not(Not(True))))
```

3. $\neg(T \wedge \neg T) \vee T$

```
let prop3 = Or(Not(And(True, Not(True))), True)
```

Вычислить значение выражения можно с помощью рекурсивной функции, аналогично арифметическому выражению:

```
let rec eval p =  
  match p with  
  | True -> true  
  | Not(prop) -> not(eval(prop))  
  | And(prop1, prop2) -> eval(prop1) && eval(prop2)  
  | Or(prop1, prop2) -> eval(prop1) || eval(prop2)
```

Функцию *eval* легко модифицировать так, чтобы она вычисляла операторы в нормальном порядке. Например, если результат вычисления первого аргумента оператора *And* равен *false*, то второй аргумент вычислять уже не нужно.

Добавим в тип данных тернарный оператор *IfThenElse*:

```

type proposition =
  | True
  | Not of proposition
  | And of proposition * proposition
  | Or of proposition * proposition
  | IfThenElse of proposition * proposition * proposition

```

Результат вычисления *IfThenElse* определяется следующим образом: если результат вычисления первого аргумента истинен, то вычислить второй аргумент. Иначе вычислить третий аргумент.

```

...
  | IfThenElse(p1, p2, p3) ->
    if eval(p1) then eval(p2) else eval(p3)

```

Теперь наша функция позволяет вычислять условные выражения:

```

eval(IfThenElse(Not(True),
  Or(True, Not(True)),
  And(True, Not(True))))
val it : bool = false

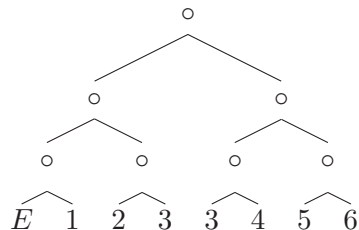
```

В тип *proposition* легко добавить поддержку переменных, получив калькулятор для логики нулевого порядка.

5 Деревья и полиморфные типы данных

Существует несколько способов описания деревьев с помощью алгебраических типов данных. Разные способы удобны для разных задач. Для простоты пока будем рассматривать только бинарные деревья из целых чисел.

Способ первый, когда данные находятся в листьях дерева:



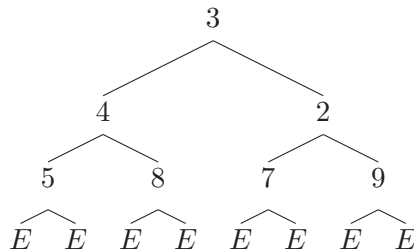
```

type Tree1 =
  | Empty
  | Leaf of int
  | Node of Tree1 * Tree1

```

Здесь *Empty* и *Leaf* – конструкторы для пустого листа и листа с числом, а *Node* – конструктор узла дерева с двумя потомками.

Способ второй, когда данные находятся в узлах дерева:



```

type Tree2 =
  | Empty
  | Node of int * Tree2 * Tree2
  
```

Здесь конструктор *Node* имеет три аргумента – значение в узле и два потомка.

В качестве примера рассмотрим работу с деревьями первого типа. Можно написать функцию, которая будет умножать на два все значения в листьях дерева. Результатом работы функции будет дерево того же типа:

```

let rec mul2 Tree =
  match Tree with
  | Empty -> Empty
  | Leaf(x) -> Leaf(x*2)
  | Node(t1, t2) -> Node(mul2 t1, mul2 t2)
  
```

Функцию можно обобщить, чтобы она применяла произвольную операцию к значениям в листьях. Фактически, это будет аналог оператора проекции *map* для списков:

```

let rec map op Tree =
  match Tree with
  | Empty -> Empty
  | Leaf(x) -> Leaf(op x)
  | Node(t1, t2) -> Node(map op t1, map op t2)
  
```

Функции *eval* для арифметических выражений и для выражений логики высказываний, по сути, являются аналогами оператора свёртки для списков.

Деревья только для целых чисел не очень полезны. *F#* позволяет обобщить этот тип данных, чтобы в листьях можно было хранить данные произвольного (но только одного!) типа. В этом случае говорят, что тип *Tree* параметризован типом *'a* (читается как *α*):


```

type 'a Tree =
  | Empty
  | Leaf of 'a
  | Node of 'a Tree * 'a Tree

```

Код функции *map* не изменится. Изменится только её тип: $map : ('a \rightarrow 'b) \rightarrow 'a\ Tree \rightarrow 'b\ Tree$. Сравните его с типом функции *map* для списков.

Можно определить тип, параметризованный более, чем одним подтипом. Синтаксис $F\#$ в этом случае будет похож на работу с обобщёнными типами в .NET. Например, тип *Rope*, чередующий значения двух разных типов, можно было бы определить как $Rope <'a, 'b>$.

Параметризованные типы ещё называют *полиморфными типами данных*.

6 Списки как алгебраический тип данных

Списки в $F\#$ – это полиморфный тип данных, который определён следующим образом:

```

type 'a list =
  | Nil
  | Cons of 'a * 'a list

```

Где *Cons* – это уже известная операция конкатенации $::$, а *Nil* – это пустой список $[]$. Таким образом, запись $[1; 2; 3]$ – это просто *синтаксический сахар* для эквивалентной записи $1 :: 2 :: 3 :: []$ или

```
Cons(1, Cons(2, Cons(3, Nil)))
```

Благодаря этому при сопоставлении с образцом можно строить сложные конструкции, например, “список, начинающийся на $[1; 2]$ ”:

```

match list with
| 1 :: 2 :: tail -> tail

```

Или “список, где первый элемент единица, а третий – тройка”:

```

match list with
| 1 :: _ :: 3 -> true

```

Если в списке интересует элемент между 1 и 3, его можно получить, связав переменную:

```

match list with
| 1 :: n :: 3 -> n

```

Такую форму сопоставления можно использовать для любых алгебраических типов данных. Например, в примере с логикой высказываний можно добавить правило, снимающее двойное отрицание перед вычислением выражения:

```
| Not(Not(e)) -> eval e
| Not(e) -> not eval e
```

Стандартный модуль F# *List* содержит ряд полезных селекторов и предикатов для списков. Например, предикат *List.isEmpty* и селекторы *List.length*, *List.head* и *List.tail*.

7 Тип необязательных значений

Предположим, что нам необходимо написать функцию деления двух значений типа *float*.

```
let div x y = x / y
```

Как определить поведение функции, если делитель – ноль? Существует несколько вариантов:

1. Вернуть специальное значение типа *float*, зарезервированное для кода ошибки. Например, *infinity* – бесконечность.
2. Вызвать исключение, “развернув” стек вызовов функций.
3. Сгенерировать системное прерывание.
4. Вызвать функцию обработки ошибки, переданную в *div* в качестве аргумента.

Каждый из подходов имеет собственные недостатки. Например, в первом случае не всегда бывает возможно зарезервировать одно из значений в качестве константы – кода ошибки. Выход – определить новый тип данных, который будет хранить или ответ типа *float*, или ошибку:

```
type floatOrError =
| Result of float
| Error
```

Очевидно, что такое определение может быть полезно и для других типов. Например, “*string* или ошибка”, “*int* или ошибка” и т.д. Эта идея воплощена во встроенном в F# типе *option*:

```
type 'a option =
| Some of 'a
| None
```

С помощью него можно переписать функцию деления *float*:

```
let div x y =
  match y with
  | 0 -> None
  | _ -> Some (x / y)
```

Её тип будет $float \rightarrow float \rightarrow float\ option$, а результат – *None*, если была попытка деления на ноль, иначе *Some* с результатом вычисления.

Для работы с *option* удобно определить селекторы *isNone* и *isSome*.

Рассмотрим функцию, которая умножает на два свой аргумент, заданный в виде типа *int option*. Она должна проверять два варианта:

```
let mulTwo arg =
  match arg with
  | None -> None
  | Some(x) -> Some(x*2)
```

Можно определить функцию, которая будет применять к аргументу произвольную операцию *op*:

```
let map op arg =
  match arg with
  | None -> None
  | Some(x) -> Some(op x)
```

Тип полученной функции: $(a \rightarrow b) \rightarrow a\ option \rightarrow b\ option$. Такое определение абсолютно аналогично определению функций *map* для типов *list* и *Tree*, описанных выше. Во всех трёх случаях функция *map* принимает в качестве второго аргумента тип, параметризованный типом *'a* и “превращает” его в тип, параметризованный другим типом – *'b* с помощью функции из *'a* в *'b*, указанной в первом аргументе.

При работе с данными, “завёрнутыми” в тип *option*, придётся либо постоянно проверять два варианта, либо в какой-то момент “достать” данные, по необходимости обработав ошибку.

Типы данных *list*, *Tree* и *option* можно рассматривать как *конструкторы типов* или *операторы типов высших порядков*, т.к. они принимают типы в качестве аргументов и возвращают новые типы в качестве результата. Типы высших порядков будут подробнее рассмотрены в следующих лекциях.

Список литературы

[Е.09] Кирпичёв Е. Элементы функциональных языков. *Журнал «Практика функционального программирования»*, 3(3), 2009.

- [P.09] Душкин Р. Алгебраические типы данных и их использование в программировании. *Журнал «Практика функционального программирования»*, 2(2), 2009.