Введение в функциональное программирование – Лекция 3. Функция как объект манипулирования. Абстракция вычисления

Олег Смирнов
oleg.smirnov@gmail.com

30 сентября 2011 г.

Содержание

Цель лекции

- Абстракция вычисления
- Функции высших порядков
- Каррирование и разрезы

1 Абстракция вычисления

В конце прошлой лекции был рассмотрен пример функции, вычисляющей квадратный корень методом Ньютона. Эта задача может быть разбита на подзадачи меньшего объёма:

1. Получить начальное приближение.
2. Итеративно повторять шаги 2 и 3.
3. Вычислить новое приближение
4. Проверить, является ли результат достаточно хорошим. Если да – вернуть ответ

В результате такой декомпозиции мы можем написать реализацию метода Ньютона, абстрагированную от конкретной задачи – от вычисления квадратного корня. Вынося, например, вычисление нового приближения в отдельную подзадачу, мы получим универсальную программу, с помощью которой можно вычислить приближение любой математической функции.

Пусть метод Ньютона принимаеть функцию вычисления нового приближения в качестве аргумента. Такая функция, манипулирующая другими функциями, называется функцией высшего порядка (higher-order function):

```
let newton first guess enough =
    let rec iter cur =
        if enough cur then cur
        else iter (guess cur)
    iter first
```

Теперь для вычисления квадратного корня достаточно реализовать несколько вспомогательных функций и передать их функции *newton* в качестве аргументов:

```
let isGood x y = abs (x - y*y) < 0.000001
let improve x y = ((x / y) + y) / 2.0
let sqrt x = newton 1. (improve x) (isGood x)
```

Вообще говоря, поддержка функций высшего порядка в языке означает способность языка манипулировать описаниями вычислений как полноценными значениями наравне с другими структурами данных. Для языков без такой способности существует ряд приёмов для имитации функций высшего порядка.

Функции высшего порядка иногда называют “операторами” или “функционалами” по аналогии с терминами функционального анализа.

2 Функции в качестве аргументов

Запишем функцию для суммирования значений i для $i = a \ldots b$:

```
let rec sum a b =
    if a > b then 0
    else a + sum (a + 1) b
```

А теперь напишем функцию для суммирования квадратов:

```
let rec sumSq a b =
    if a > b then 0
    else a*a + sumSq (a + 1) b
```

Обе функции могут быть переписаны в виде итеративного процесса, но в этом примере это не важно. Код функций похож, как будто он написан с использованием общего шаблона. Присутствие такого шаблона является веским доводом в пользу того, что здесь скрыта полезная абстракция. Действительно, в математической нотации эта абстракция называется “сигма-записью”:

$$sum = \sum_{i=a}^{b} i$$

$$sumSq = \sum_{i=a}^{b} i^2$$

Сила сигма-записи в том, что она выражает само понятие суммы вообще, а не просто некоторые конкретные суммы. Выразим эту идею в виде функции:

```
let rec sumOf op a next b =
    if a > b then 0
    else (op a) + sumOf op (next a) next b
```

Здесь функция next означает “приращение аргумента”. Её можно использовать, например, при вычислении суммы ряда

$$\frac{\pi}{4} = \frac{1}{3 \cdot 5} + \frac{1}{7 \cdot 9} + \frac{1}{9 \cdot 11} + \dots$$

Выразим sum и sumSq через sumOf:

```
let id a = a
let inc a = a + 1
let square a = a*a
let sum a b = sumOf id a inc b
let sumSq a b = sumOf square a inc b
```

Такая запись на порядок понятней. Теперь sumOf выражает идею суммирования, а sum и sumSq – используют эту идею. Перепишем функцию sumOf в виде рекурсивного процесса:

```
let sumOf op a next b =
    let rec sumIter a s =
        if a > b then s
        else sumIter (next a) (s + (op a))
    sumIter a 0
```

Функции sum и sumSq автоматически стали итеративными процессами! Абстракция вычислений позволяет “спрятать” обобщённый вычислительный процесс и изменять его реализацию по мере надобности. Кроме того, идею суммы можно ещё сильнее обобщить, например, добавить предикат для “отфильтровывания” некоторых термов, обобщить условие останова и т.д.

Если вычислить сумму $\frac{1}{n}$ в интервале от 1 до 100000000, то получим ответ 15.403683. Это отличается от правильного ответа 18.997896 на 3.5! Причина такой потери точности – разрядность типа float. Если сложить 10^6 и 10^{-6} типа float, то результат будет 10^6, т.к. тип float не обладает точностью в 12 разрядов. Если к 10^6 прибавить 10^{12} раз по 10^{-6}, то результат всё равно будет 10^6, хотя правильный ответ $2 \cdot 10^6$.

Способ суммирования тем оптимальнее, чем реже мы складываем большое число с маленьким. Поэтому на первом шаге нужно сложить два самых маленьких числа, затем сложить оставшиеся числа и результат предыдущей операции и т.д. Это та же самая задача – суммирование большого списка чисел, просто список изменился. Алгоритм можно эффективно реализовать в виде итеративного процесса с использованием

приоритетной очереди, затем алгоритм встроить в функцию sumOf. Возможен ряд дополнительных оптимизаций, например переключение между “стандартным алгоритмом” и “алгоритмом с очередью” в зависимости от значения аргументов.

Лямбда-функции

До сих пор мы явно декларировали функции, передаваемые в sumOf в качестве параметров. Однако это не обязательное действие. Функцию можно определить в виде выражения, не связывая с ним никакое имя. В синтаксисе F# для этого используется ключевое слово fun и конструкция $fun\ \langle arg \rangle\ \rightarrow\ \langle body \rangle$. Например, функцию sum можно было определить так:

```
let sum a b = sumOf (fun x -> x) a (fun x -> x + 1) b
```

Такая нотация называется определением лямбда-функции или анонимной функции – функцией без имени.

Операции над списками

Списком в F# называется упорядоченная последовательность значений одного типа. Например, список значений типа *int* можно определить как:

```
let listInts = [1; 2; 3; 100]
```

Список значений типа *bool*:

```
let listBools = [false; true; true; true]
```

Пустой список обозначается как []. Для списков определена операция ::, позволяющая присоединить элемент к “голове” списка:

```
0 :: [3; 2; 1]
val it : int list = [0; 3; 2; 1]
```

Два списка можно “склеить” в новый список операцией @:

```
[true; true] @ [false; true; false]
val it : bool list = [true; true; false; true; false]
```

Список в F# неизменяем в том смысле, что в нём невозможно изменить элемент.

Для работы со списками удобно использовать сопоставление с образцом:

```
match someList with
| head :: tail -> tail
| [] -> []
```

Такая операция сначала сравнит список *someList* с образцом *head* :: *tail*. Если это возможно, и список представим в виде конкатенации (склейки) головного элемента *head* и списка из остальных элементов (хвоста) *tail*, то выполнится первая ветка и результатом операции будет список-хвост *tail*. Если же нет, и список *someList* не содержит элементов (т.е. совпадает с []), результатом операции будет пустой список.

Напишем функцию суммирования списка в рекурсивном и итеративном варианте:

```
let rec sum list =
    match list with
    | head :: tail -> head + sum tail
    | [] -> 0

let sum list =
    let rec loop list acc =
        match list with
        | head :: tail -> loop tail (acc + head)
        | [] -> acc
    loop list 0
```

Аналогично суммированию чисел в интервале, здесь можно выделить общую идею – "свёртку" списка с помощью некоторой бинарной операции *op* и начального значения *base*. Функция свёртки применяет *op* к двум аргументам – к *base* и к первому элементу списка x_1. На следующем шаге *op* применяется к результату предыдущего действия и элементу x_2 и так далее, в результате вычисляя формулу:

$$op\ (\ldots(op\ (op\ base\ x_1)\ x_2)\ldots)\ x_n$$

В функциональном программировании эту операцию обычно называют $fold$. Различают "левую" и "правую" свертки ($foldl$ и $foldr$) или же "свёртку вперёд" и "свёртку назад" ($fold$ и $foldBack$). Разница в том, что правая или обратная свёртка вычисляет формулу

$$op\ x_1\ (op\ x_2\ (\ldots(op\ x_n\ base)\ldots))$$

Результаты левой и правой свёртки будут совпадать, если операция *op* обладает свойством ассоциативности.

Пусть функция свёртки опредена как $fold\ op\ base\ list$. Тогда на её основе можно определить ряд полезных операций над списками. Например, суммирование элементов списка:

```
let sumList list = fold (fun a b -> a + b) 0 list
```

Минимум и максимум в списке:

```
let min a b =
    if a < b then a
    else b
let max a b =
    if a > b then a
    else b
let minList list = fold min infinity list
let maxList list = fold max (-infinity) list
```

Функции *all* и *any* для списков типа *bool*:

```
let all list = fold (fun a b -> a && b) true list
let any list = fold (fun a b -> a || b) false list
```

3 Функции в качестве результата

Функции высшего порядка могут не только принимать другие функции в качестве аргумента, но и возвращать их в качестве результата.

Определим несколько полезных операторов:

- Тождественный оператор:

  ```
  let id f x = f x
  ```

- Оператор композиции функций:

  ```
  let inline (<<) f g x = f (g x)
  ```

- Обратная композиция функций:

  ```
  let inline (>>) f g x = g (f x)
  ```

- Оператор применения функции:

  ```
  let inline ($) f x = f x
  ```

- Оператор смены порядка аргументов *flip*:

  ```
  let flip f x y = f y x
  ```

- Оператор вычисления в заданной точке *at*:

  ```
  let inline (|>) x f = f x
  ```

Другое определение:

```
let at x f = (flip ($)) x f
```

Ключевое слово *inline* в определении композиции функций позволяет использовать этот оператор в инфиксной форме, т.е. ставит его между аргументами. Оператор обратной композиции часто бывает удобней, т.к. он применяет функции в том порядке, в котором они записаны:

```
let incAndSquare x = (inc >> square) x
```

С помощью оператора вычисления в заданной точке удобно строить конвеерную обработку, значительно упрощая вид функции. Сравним:

```
let complexFun x = square (add 5 (cube x))
let complexFun x = x |> cube |> add 5 |> square
```

- Оператор отображения списка *map* определяется следующим образом:

```
let rec map op list =
    match list with
    | head :: tail -> op head :: map op tail
    | [] -> []
```

- Оператор фильтрации списка *filter*:

```
let rec filter pred list =
    match list with
    | head :: tail -> if pred head then
                          head :: filter pred tail
                      else filter pred tail
    | [] -> []
```

Теперь, комбинируя функции с помощью оператора композиции, легко построить потоковую обработку списка. Например, функция, которая возводит каждый элемент списка в четвёртую степень, предварительно отбросив отрицательные элементы:

```
let filterNegAndQuad list =
    list |> filter (fun x -> x > 0) |> map (square >> square)
```

Очевидно, что такой код гораздо более читаем, чем “лесенка” из вложенных циклов в императивном стиле.

На практике функции высшего порядка имеют множество применений. Например, создание подпроцесса (нити, потока), реализация обобщённых алгоритмов (например, сортировки, куда передаётся функция сравнения элементов) и передача функции обратного вызова (callback) в событийно-ориентированном программировании (event-driven programming).

4 Каррирование и разрезы

В предыдущем примере был построен “конвеер” из оператора фильтрации и оператора отображения списка, причём в каждом из них первый аргумент был “зафиксирован”. Для фильтра был зафиксирован предикат $fun\ x\ \rightarrow\ x > 0$, а для отображения – функция возведения в четвёртую степень $square\ >>\ square$.

Такой приём называется “каррингом” или частичным вычислением функции. По определению, частичным применением n-арной функции f называется конструкция, значением которой является $(n-k)$-арная функция, соответствующая f при фиксированных значениях некоторых k из n её аргументов.

На практике имеет смысл упорядочивать аргументы функции по мере наростания “изменчивости”: вначале идут “аргументы-опции”, а затем “аргументы- данные”. Например, в описанном выше операторе $filter$ первым идёт параметр-предикат $pred$, а затем список $list$. С точки зрения программиста, удобно зафиксировать фильтрацию с предикатом в виде функции от одного аргумента, а затем применять её к различным спискам.

Некоторые языки позволяют превратить любую функцию от двух аргументов в бинарный оператор. Фиксация левого или правого аргумента такого бинарного оператора называется левым и правым разрезом функции соответственно.

5 Бесточечный стиль

В бесточечном стиле программирования мы опускаем аргументы (точки применения) функции в левой и в правой частях ее определения. То есть вместо

```
sum list = fold add 0 list
```

в бесточечном стиле мы пишем

```
sum = fold add 0
```

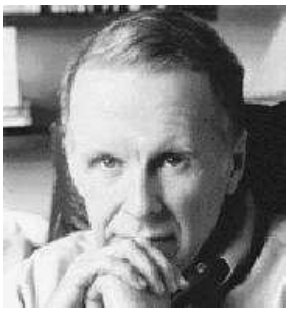
Такое отбрасывание аргументов возможно, только если самый правый аргумент в левой части является самым правым и в правой части (и при этом больше нигде в правой не используется). В нашем примере это аргумент $list$.

В бесточечном стиле функции не определяются через результат вычислений над аргументами, а выражаются через другие функции при помощи операторов композиции функции и подобных. Такой стиль часто

позволяет уменьшить количество лишней информации в определении, делая его более лаконичным и читабельным.

Понятие карринга было впервые введено русским математиком Шейнфинкелем в 1924 г. при создании комбинаторной логики. Хаскелл Карри упоминал его в своих работах как “первооткрывателя”. Бесточечный стиль в современном понимании этого слова был впервые описан Джоном Бэкусом в его лекции “Can programming be liberated from the Von Neumann style?” [Bac78] прочитанной на церемонии вручения премии Тьюринга в 1977 году.

Моисей Исаевич Шейнфинкель
(1889 - 1942)



John Backus
(1924 - 2007)

Обычно в языках программирования существуют ограничения на способы, с помощью которых можно манипулировать элементами вычисления. Элементы или значения, на которые накладывается наименьшее число ограничений, ещё называются “полноправными” или элементами первого класса (first-class). Вот некоторые из их “прав и привилегий”:

- Их можно называть с помощью переменных.
- Их можно передавать в процедуры в качестве аргументов.
- Их можно возвращать из процедур в виде результата.
- Их можно включать в структуры данных

Список литературы

[Bac78] J. Backus. Can Programming be Liberated from the Von Neumann Style? *Communications of the ACM*, 21(8):613–641, 1978.

[В.09] Брагилевский В. Пределы выразительности свёрток. *Журнал «Практика функционального программирования»*, 3(3), 2009.

[Д.10] Москвин Д. Сечения композиции как инструмент бесточечного стиля. *Журнал «Практика функциональъного программирования»*, 4(4), 2010.

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