Введение в функциональное программирование – Лекция 1. Язык F#. Порядок применения функции

Олег Смирнов
oleg.smirnov@gmail.com

16 сентября 2011 г.

Содержание

Цель лекции

- Базовые понятия языка F#
- Нормальный и аппликативный порядок применения функции
- Проверка и вывод типов

1 Язык F#

F# – это мультипарадигменный язык программирования, разработанный в Microsoft Research и предназначенный для исполнения на платформе Microsoft .NET. F# наследует от языков OCaml и Haskell и сочетает функциональную парадигму с императивными возможностями и объектной моделью .NET.

История языка началась в 2002-м году в исследовательской группе под руководством Дона Сайма в Кэмбридже. В 2005-м вышла первая версия F#, а уже в 2010-м он был включён в поставку Microsoft Visual Studio 2010.

Базовый элемент синтаксиса F# – это *выражение*. Примеры выражений:

```
11
3 + 5 + 3
(1 + 2) * (3 + 4)
"Hello"
"Hello" + "␣World"
intToString 5
"Hello␣World" + 1
```

Результатом вычисления выражения является *значение*. Значение будем отделять от выражения символом =>:

```
11 => 11
3 + 5 + 3 => 11
(1 + 2) * (3 + 4) => 21
"Hello" => "Hello"
"Hello" + "␣World" => "Hello␣World"
intToString 5 => "5"
"Hello␣World" + 1 – ошибка!
```

Значение получается из выражения в процессе *вычисления*. Будем обозначать один шаг вычисления символом |->. Например:

```
    (1 + 2) * (3 + 4)
|-> 3 * (3 + 4)
```

```
|-> 3 * 7
|-> 21
```

Вычисления могут происходить последовательно, как в предыдущем примере, или параллельно. Во втором случае, независимые друг от друга выражения вычисляются одновременно:

```
    (1 + 2) * (3 + 4)
|-> 3 * 7
|-> 21
```

2 Типы

Значение "$Hello\ World$" + 1 не будет вычислено потому, что оно *некорректно типизировано* (ill-typed). В F# каждое выражение имеет *тип*. Тип выражения говорит о том, какие значения могут получиться при вычислении этого выражения и получатся ли какие-нибудь значения вообще. Выражение *корректно типизировано* (well-typed), если у него есть как минимум один тип. У выражение может быть более одного типа – это называется *полиморфизм*. Хорошее описание полиморфизма и его разновидностей есть в статье Люка Карделли "On understanding types..." [CW85].

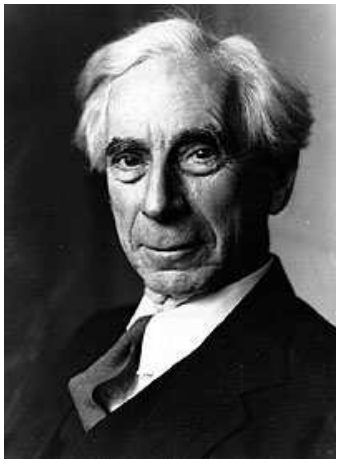
Теория типов базируется на идеях Б. Рассела и А. Уайтхеда из работы "Principia Mathematica" [WR27]. Они предложили сопоставлять высказыванию (пропозициональной функции) некоторый тип, который является его доменом или областью определения. Иерархия, составленная из таких типов, должна была разрешить парадоксы наивной теории множеств и послужить основанием математики.

Будем записывать тип выражения в формате $\langle exp \rangle : \langle type \rangle$. Например:

```
11 : int
"Hello" : string
"Hello World" + 1 - ошибка!
```

В общем случае, тип задаётся набором *значений*, которые могут принимать выражения этого типа и набором *операций* над типом. Пример:

- Тип *int*
  - Значения: 11, 23, ~42, ...
  - Операции: +, -, $intToString$, ...
- Тип *string*



Bertrand Russell
(1872 - 1970)

Alfred North Whitehead
(1861 - 1947)

  - Значения: "$Hello$", "$World$", ...
  - Операции: +, size, ...

- Тип $float$
  - Значения: 0.0, 2.78, 3.14, ...
  - Операции: +, -, /, *, ... (N.B.: имена совпадают с операциями типа int, но это другие операции)

Аналогично C#, базовые типы являются псевдонимами для встроенных типов .NET: System.Int32, System.String и т.д.

3 Декларации

Программа на F# состоит из *деклараций*. Рассмотрим декларацию *переменных*:

```
let x : int = 3 + 5 + 3
```

Переменной с именем x будет сопоставлено значение выражения $3+5+3$ (типа int). В общем виде переменные объявляются конструкцией $let\ \langle var\rangle\ :\ \langle type\rangle\ =\ \langle exp\rangle$.

При вычислении последовательности деклараций let, вычисляется первая декларация, а затем её значение *подставляется* в следующие выражения, в места, где упоминается соответствующая переменная. Например:

```
let x : int = 2 + 3
let y : int = x + 1
let z : int = 5 + y
```

Шаг 1:

```
let x : int = 5
let y : int = x + 1
let z : int = 5 + y
```

Шаг 2:

```
let x : int = 5
let y : int = 6
let z : int = 5 + y
```

Шаг 3:

```
let x : int = 5
let y : int = 6
let z : int = 11
```

Что произойдёт, если написать две декларации с одинаковым именем переменной?

```
let x : int = 5
let x : int = 6
```

В отличии от переменных в императивной парадигме, в функциональном мире переменные не меняют своего значения. Их скорее следовало бы называть "постоянными". Имя переменной *связывается* с выражением (var binding).

Таким образом, во второй строке этого примера будет создано новое связывание между именем x и выражением 6, а старое будет "забыто".

Второй тип деклараций – это декларация типов:

```
type coordinate = int*int
```

После этой декларации *переменная типа coordinate* в программе будет означать $int*int$. Общая форма декларации нового типа: $\langle tyvar \rangle = \langle type \rangle$.

Нотация $int * int$ здесь означает пару из двух int. Её также можно записать как (int, int).

Область видимости в F# – *лексическая* (она же статическая).

4 Проверка и вывод типов

F# – статически типизированный язык с поддержкой определения типов. Это означает, что все конструкции имеют известный тип уже на этапе компиляции, а не на этапе выполнения программы.

Алгоритм проверки типов упрощённо похож на вывод доказательства, где типы элементарных выражений – это аксиомы, а операции – это теоремы (правила вывода). Например, аксиомы:

```
23 : int
"Hello" : string
```

Правила вывода:

```
<exp1> + <exp2> : int если <exp1> : int и <exp2> : int
<exp1> + <exp2> : string если <exp1> : string и <exp2> : string
```

Тогда $5 + 3 * 2 : int$, т.к. $5 : int$ – аксиома и $3 * 2 : int$, т.к. $3 : int$ и $2 : int$ – аксиомы.

Кроме того, F# – язык со строгой типизацией. Это означает, что в выражениях отсутствует неявное приведение типов. Например, выражение $0.1 + 1$ вызовет ошибку компиляции, т.к. операция + определена для двух *int* или двух $float$, но не для $float$ и *int*.

5 Функции

Как и в других языках, функция в языке F# имеет имя, может иметь параметры и принимать аргументы, а также функция имеет тело. В языке F# функции являются *объектами первого порядка*, т.е. могут возвращать функции и принимать функции в качестве аргумента.

Функции определяются с помощью ключевого слова *let*. Формат декларации в общем виде: $let\langle func\rangle\ \langle arg\rangle\ =\ \langle body\rangle$. Например, функцию $f(x) = 3 * x + 2$ можно определить как:

```
let f (x : int) : int = (3*x) + 2
```

В случае, когда функции необходимо рекурсивно вызывать себя, в определение добавляется ключевое слово *rec*. Пример – функция вычисления факториала:

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1);;
```

6 Функциональный тип

Тип функции определяется как $\langle type1 \rangle \to \langle type2 \rangle$, где $\langle type1 \rangle$ – это тип аргумента, а $\langle type2 \rangle$ – тип возвращаемого значения. Например, тип функции из первого примера будет $int \to int$.

Как быть с функцией, у которой количество аргументов превышает единицу? Например, функция суммы:

```
let sum a b = a + b
```

Её тип будет $int \to int \to int$ или же $int \to (int \to int)$. Таким образом исходную функцию можно рассматривать как функцию от одного параметра типа int, которая возвращает другую функцию типа $int \to int$. Всякая функция с более чем одним аргументом может интерпретироваться как функция от первого аргумента, возвращающая функцию от оставшихся при фиксированном значении первого.

Этот приём функционального программирования называется *каррированием* или *каррингом* в честь математика и логика Хаскелла Карри.

Haskell Curry
(1900 - 1982)

Благодаря выводу типов, тип функции и типы аргументов во многих случаях могут быть опущены.

7 Применение и его порядок

Главная операция для функции – это *применение* – подстановка аргументов в тело функции:

```
    f 3
|-> (3 * 2) + 2
|-> 6 + 2
|-> 8
```

Есть несколько вариантов порядка применения:

- аппликативный – самый "естественный" порядок, при котором аргументы функции полностью вычисляются до её вызова. Такой порядок используется в большинстве языков программирования
- нормальный – порядок, при котором аргументы функции начинают вычисляться только тогда, когда в них возникает необходимость
- ленивый – вариант реализации нормального порядка, при котором не происходит дублирования вычислений

Для примера определим функцию *double*:

```
let double x = x + x
```

Пример аппликативного порядка:

```
    double (double 5)
|-> double (5 + 5)
|-> double 10
|-> 10 + 10
|-> 20
```

Пример нормального порядка:

```
    double (double 5)
|-> (double 5) + (double 5)
|-> 10 + 10
|-> 20
```

В F# есть поддержка ленивого порядка редукции с помощью ключевого слова *lazy*. Порядки редукции и ленивые вычисления будут подробно рассмотрены во втором семестре курса.

Список литературы

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17:471–523, December 1985.

[WR27] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1925–1927.

[Д.10] Лазин Е. Моисеев М. Сорокин Д. Введение в F#. *Журнал «Практика функционального программирования»*, 5(5), 2010.

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