Функциональное программирование на языке F#

Юрий Литвинов

19.02.2016г

О чём этот курс

- Теория и практика функционального программирования
  - λ-исчисление
  - Базовые принципы ФП (программирование без состояний, функции высших порядков, каррирование и т.д.)
  - Типы в функциональном программировании (немутабельные коллекции, генерики, автообобщение и т.д.)
  - Паттерны функционального программирования (CPS, монады, point-free)
- Программирование на F#
  - ООП в F#
  - Асинхронное и многопоточное программирование

Отчётность

- Домашка (довольно много)
- Одна контрольная в середине семестра
- Курсовая работа
- Доклад (-1 домашка)

Отступление про курсовые работы

- Курсовая по дисциплине, отдельно в зачётку не идёт
- Семестровая + некоторая наука + текст
- Объём — 5-7 страниц содержательного текста
- Конференции
  - СПИСОК-2016 — 26 апреля
  - «Современные технологии в теории и практике программирования» — 20 марта
  - SEIM-2016 — 10 марта
  - SYRCoSE — 1 апреля

Структура отчёта

- Титульный лист (http://math.spbu.ru/rus/study/alumni_info.html)
- Оглавление
- Введение в предметную область, постановка задачи
- Обзор литературы и существующих решений
- Описание предлагаемого решения, сравнение с существующими
- Заключение
- Список источников (ГОСТ Р 7.0.5–2008)
- Приложения (если есть)

Где брать темы

- Продолжать начатое
- Студпроекты Теркома
  - 25 февраля в 12:50 в ауд. 405
- Придумать самим
  - Политически немудро, но может быть интересно
- Взять что-нибудь новое у меня
  - GUI для метамоделирования на лету
  - Автораскладывалка элементов
  - Плагин к Qt Creator

Императивное программирование

Программа как последовательность **операторов**, изменяющих **состояние** вычислителя.
Для конечных программ есть **начальное состояние**, **конечное состояние** и последовательность переходов:

$$\sigma = \sigma_1 \rightarrow \sigma_2 \rightarrow ... \rightarrow \sigma_n = \sigma'$$

Основные понятия:

- Переменная
- Присваивание
- Поток управления
    - Последовательное исполнение
    - Ветвления
    - Циклы

Функциональное программирование

Программа как вычисление значения **выражения** в математическом смысле на некоторых входных данных.

$$\sigma' = f(\sigma)$$

- Нет состояния $\Rightarrow$ нет переменных
- Нет переменных $\Rightarrow$ нет циклов
- Нет явной спецификации потока управления

Порядок вычислений не важен, потому что нет состояния, результат вычисления зависит только от входных данных.

Сравним

C++

```
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

F#

```
let rec factorial x =
    if x = 1 then 1 else x * factorial (x - 1)
```

Как с этим жить

- Состояние и переменные «эмулируются» параметрами функций
- Циклы «эмулируются» рекурсией
- Последовательность вычислений — рекурсия + параметры

F#

```
let rec sumFirst3 ls acc i =
    if i = 3 then
        acc
    else
        sumFirst3
            (List.tail ls)
            (acc + ls.Head)
            (i + 1)
```

Зачем

- Строгая математическая основа
- Семантика программ более естественна
  - Применима математическая интуиция
- Программы проще для анализа
  - Автоматический вывод типов
  - Оптимизации
- Более декларативно
  - Ленивость
  - Распараллеливание
- Модульность и переиспользуемость
- Программы более выразительны

Пример: функции высших порядков

F#

```
let sumFirst3 ls =
    Seq.fold
        (fun x acc -> acc + x)
        0
        (Seq.take 3 ls)
```

F#

```
let sumFirst3 ls = ls |> Seq.take 3 |> Seq.fold (+) 0
```

F#

```
let sumFirst3 = Seq.take 3 >> Seq.fold (+) 0
```

Ещё пример

Возвести в квадрат и сложить все чётные числа в списке

F#

```
let calculate =
    Seq.filter (fun x -> x % 2 = 0)
    >> Seq.map (fun x -> x * x)
    >> Seq.reduce (+)
```

Почему тогда все не пишут функционально

- Чистые функции не могут оказывать влияние на внешний мир. Ввод-вывод, работа с данными, вообще выполнение каких-либо действий не укладывается в функциональную модель.
- Сложно анализировать производительность, иногда функциональные программы проигрывают в производительности императивным. «Железо», грубо говоря, представляет собой реализацию машины Тьюринга, тогда как функциональные программы определяются над λ-исчислением.
- Требуется математический склад ума и вообще желание думать.

Лямбда-исчисление

Математическая основа функционального программирования

- Формальная система, основанная на λ-нотации, ещё одна формализация понятия «вычисление», помимо машин Тьюринга (и нормальных алгорифмов Маркова, если кто-то про них помнит)
- Введено Алонзо Чёрчем в 1930-х для исследований в теории вычислимости
- Имеет много разных модификаций, включая «чистое» λ-исчисление и разные типизированные λ-исчисления
- Реализовано в языке LISP, с тех пор прочно вошло в программистский обиход (даже анонимные делегаты в C# называют лямбда-функциями, как вы помните)

Лямбда-нотация

Способ вводить функции, не придумывая для них название каждый раз

$$x \to t[x] \Longrightarrow \lambda x.t[x]$$

Например,

$$\lambda x.x$$

$$\lambda x.x^2$$

Применение функции (или аппликация)

Математически привычно

$$f(x)$$

Но непонятно, о чём идёт речь — о функции f, принимающей аргумент x, или о результате применения f к x.
В лямбда-исчислении $f(x)$ обозначается как

$$f\ x$$

При этом принято, что

$$\lambda x.x\ y = \lambda x.(x + y), \quad \lambda x.x\ y \neq (\lambda x.x) + y$$

Примеры записи:

$$(\lambda x.x^2)\ 5 = 25$$

$$(\lambda x.\lambda y.x + y)\ 2\ 5 = 7$$

Каррирование (Currying)

В λ-исчислении не нужны функции нескольких переменных:

$$\lambda x.\lambda y.x + y \stackrel{def}{=} \lambda x\, y.x + y$$

Можно понимать как функцию, которая возвращает функцию:

$$\lambda x.\lambda y.x + y \equiv \lambda x.(\lambda y.x + y)$$

$$\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$$

Частичное применение:

$$(\lambda x.\lambda y.x + y)\ 5 \equiv \lambda x.(x + 5)$$

λ-исчисление как формальная система

Внезапно, математика на парах по проге

Всё, что было выше, хорошо, но неформально. За нотацией должен стоять чёткий синтаксис и семантика.

Нетипизированное лямбда-исчисление:

- Всё — λ-термы (числа и операции вводятся через них)
  - Не делается различий между данными и функциями, можно применять функцию к функции
- Процесс вычисления вводится как набор формальных преобразований над λ-термами
  - **Операционная** семантика

λ-термы

λ-терм — это:

- Переменная: $v \in V$, где V — некоторое множество, называемое множеством переменных
- Аппликация: если A и B — λ-термы, то $A\ B$ — λ-терм.
- λ-абстракция: если A — λ-терм, а v — переменная, то $\lambda v.A$ — λ-терм
- Других способов получить λ-терм нет

Соглашения об ассоциативности

Чтобы не надо было писать кучу скобок

- Аппликация левоассоциативна: $F\ X\ Y = (F\ X)\ Y$
- λ-абстракция правоассоциативна: $\lambda x\ y.M = \lambda x.(\lambda y.M)$
- λ-абстракция распространяется вправо настолько, насколько возможно: $\lambda x.M\ N = (\lambda x.M\ N)$

Свободные и связанные переменные

- λ-абстракция $\lambda x.T[x]$ **связывает** переменную x в терме $T[x]$
- Если значение выражения зависит от значения переменной, то говорят, что переменная **свободно** входит в выражение

Пример:

$$\sum_{m=1}^{n} m = \frac{n(n+1)}{2}$$

Здесь n входит свободно, а m связана. Имя связанной переменной можно менять:

$$\int_0^x 2y + a\,dy = x^2 + ax \longrightarrow \int_0^x 2z + a\,dz = x^2 + ax$$

но

$$\int_0^x 2a + a\,da \neq x^2 + ax$$

Свободные и связанные переменные, формально

Как обычно, определение рекурсивно по структуре терма:

- $FV(x) = x$
- $FV(ST) = FV(S) \cup FV(T)$
- $FV(\lambda x.S) = FV(S) \setminus \{x\}$

- $BV(x) = \emptyset$
- $BV(ST) = BV(S) \cup BV(T)$
- $BV(\lambda x.S) = BV(S) \cup \{x\}$

Примеры:

$$S = (\lambda x\, y.x)(\lambda x.z\, x) \Rightarrow FV(S) = z, BV(S) = \{x, y\}$$

Подстановка

$T[x := S]$ - подстановка в терме T терма S вместо всех свободных вхождений переменной x (например, $x[x := T] = T$).
Проблема:

$$(\lambda y.x + y)[x := y] = \lambda y.y + y$$

Решения:

- Запретить свободным переменным иметь одинаковые имена и называться так же, как связанные (соглашение Барендрегта)
- Переименовывать связанные переменные «на лету» перед выполнением подстановки

Подстановка, формально

- $x[x := T] = T$
- $y[x := T] = y$
- $(S_1\ S_2)[x := T] = S_1[x := T]\ S_2[x := T]$
- $(\lambda x.S)[x := z] = \lambda x.S$
- $(\lambda y.S)[x := T] = \lambda y.(S[x := T])$, если $y \notin FV(T)$ или $x \notin FV(S)$
- $(\lambda y.S)[x := T] = \lambda z.(S[y := z][x := T])$, иначе ($z$ при этом выбирается так, что $z \notin FV(S) \cup FV(T)$

Зачем мы это делали

Можно ввести отношение **равенства** над термами, имеющее физический смысл «термы означают одно и то же» и отношение **редукции**, означающее «термы имеют одинаковое **значение**», что нужно для определения **вычисления** (хотя заметьте, что пока в формальной системе даже понятия «значение» нет).
Делать это мы будем, определив аксиомы и правила вывода над термами, через **преобразования** термов.

Преобразования

α-преобразование : $\lambda x.S \to_\alpha \lambda y.S[x := y]$ при условии, что $y \notin FV(S)$. Даёт возможность переименовывать связанные переменные.

β-преобразование : $(\lambda x.S)\,T \to_\beta S[x := T]$. Определяет процесс вычисления.

η-преобразование : $\lambda x.T\,x \to_\eta T$, если $x \notin FV(T)$. Обеспечивает **экстенсиональность** — две функции экстенсионально эквивалентны, если на всех одинаковых входных данных дают одинаковый результат:

$$\forall x\, F\, x = G\, x$$

Аксиомы равенства λ-термов

$$\frac{S \rightarrow_\alpha T \text{ или } S \rightarrow_\beta T \text{ или } S \rightarrow_\eta T}{S = T}$$

$$\frac{}{T = T}$$

$$\frac{S = T}{T = S}$$

$$\frac{S = T \wedge T = U}{S = U}$$

$$\frac{S = T}{S\,U = T\,U}$$

$$\frac{S = T}{U\,S = U\,T}$$

$$\frac{S = T}{\lambda x.S = \lambda x.T}$$

Вычисление, что мы хотим

Очевидно, что равенство — это отношение эквивалентности. Оно «не даёт терять информацию», потому что всегда можно вернуться к исходному терму. А мы хотим вычислять значение терма, то есть всё-таки терять информацию о синтаксисе терма, сохраняя его «смысл». Так что уберём симметричность, получив отношение **β-редукции**, которое уже не эквивалентность и позволяет делать с термом что-то осмысленное.

Аксиомы β-редукции

$$\frac{S \to_\alpha T \text{ или } S \to_\beta T \text{ или } S \to_\eta T}{S \to_\beta T}$$

$$\frac{}{T \to_\beta T}$$

$$\frac{S \to_\beta T \wedge T \to_\beta U}{S \to_\beta U}$$

$$\frac{S \to_\beta T}{S\,U \to_\beta T\,U}$$

$$\frac{S \to_\beta T}{U\,S \to_\beta U\,T}$$

$$\frac{S \to_\beta T}{\lambda x.S \to_\beta \lambda x.T}$$

Пример

Редукция не всегда уменьшает размер терма

$$(\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x) \to_\beta$$

$$(\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x) \to_\beta$$

$$(\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x) \to_\beta \ldots$$

так что

$$(\lambda x.y)\;((\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)) \to_\beta$$

$$(\lambda x.y)\;((\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)) \to_\beta$$

$$(\lambda x.y)\;((\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)) \to_\beta \ldots$$

но

$$(\lambda x.y)\;((\lambda x.x\,x\,x)\;(\lambda x.x\,x\,x)) \to_\beta y$$