Элементы функционального программирования Часть 2

Д.Н. Лавров
2017

Что такое функциональное программирование?

- **Функциональное программирование** – это стиль программирования, использующий только композиции функций. Другими словами, это программирование в выражениях, а не в императивных командах.

Что такое функциональное программирование?

- Дэвид Мертц (Functional Programming in Python):
  - "Наличие функций первого класса (функции наравне с другими объектами можно передавать внутрь функций).
  - Рекурсия является основной управляющей структурой в программе.
  - Обработка списков (последовательностей).
  - Запрещение побочных эффектов у функций, что в первую очередь означает отсутствие присваивания (в "чистых" функциональных языках)
  - Запрещение операторов, основной упор делается на выражения. Вместо операторов вся программа в идеале одно выражение с сопутствующими определениями.
  - Ключевой вопрос: что нужно вычислить, а не как.
  - Использование функций более высоких порядков (функции над функциями над
  - функциями)".

Функциональная программа

- В математике функция отображает объекты из одного множества (множества определения функции) в другое (множество значений функции).
- Математические функции (их называют чистыми) однозначно вычисляют результат по заданным аргументам. Чистые функции не должны хранить в себе какие-либо данные между двумя вызовами.
- Программы в функциональном стиле конструируются как **композиция функций**.
- ***Лямбда-исчисление*** является основой для функционального программирования, многие функциональные языки можно рассматривать как «надстройку» над ними

Lambda-исчисление

- История вопроса
- Проблема разрешения — задача из области оснований математики, сформулированная Давидом Гильбертом в 1928 году: найти алгоритм, который бы принимал в качестве входных данных описание любой проблемы разрешимости (формального языка и математического утверждения «S» на этом языке) — и, после конечного числа шагов, останавливался бы и выдавал один из двух ответов: «Истина» или «Ложь», — в зависимости от того, истинно или ложно утверждение «S».

Lambda-исчисление

- В 1936 году — Алонзо Чёрч и независимо от него Алан Тьюринг опубликовали работы, в которых показали, что не существует алгоритма для определения истинности утверждений арифметики, а поэтому и более общая проблема разрешения также не имеет решения. Этот результат получил название: «теорема Чёрча — Тьюринга».

Lambda-исчисление

- в середине 60-х Питер Ландин не отметил, что сложный язык программирования проще изучать, сформулировав его ядро в виде небольшого базового исчисления, выражающего самые существенные механизмы языка и дополненного набором удобных производных форм, поведение которых можно выразить путем перевода на язык базового исчисления. В качестве такой основы Ландин использовал лямбда-исчисление Чёрча.

Lambda-исчисление

- формальная система, разработанная американским математиком Алонзо Чёрчем, для формализации и анализа понятия вычислимости.
- В основу λ-исчисления положены две фундаментальные операции:
  - Аппликация
  - Абстракция

Аппликация

- **Аппликация** (лат. applicatio — прикладывание, присоединение) означает применение или вызов функции по отношению к заданному значению.
- Её обычно обозначают **f a**, где **f** — функция, а **a** — аргумент. Это соответствует общепринятой в математике записи **f(a)**, однако для λ-исчисления важно то, что **f** трактуется как алгоритм, вычисляющий результат по заданному входному значению.
- Двойная трактовка.

Абстракция

- **Абстракция** или **λ-абстракция** (лат. abstractio — отвлечение, отделение) в свою очередь строит функции по заданным выражениям.
- Если **t** ≡ **t** [**x**] — выражение, свободно содержащее **x**, тогда запись **λ x**.**t** [**x**] означает: λ-функция от аргумента **x**, которая имеет вид **t** [**x**], обозначает функцию **x** ↦ **t** [**x**]
- Сравните с lambda **x**: **t**(**x**)
- С помощью абстракции можно конструировать новые функции.

α-эквивалентность

- Основная форма эквивалентности, определяемая в лямбда-термах, это альфа-эквивалентность.
- Например, **λ x** . **x** и **λ y** . **y** альфа-эквивалентные лямбда-термы и оба представляют одну и ту же функцию (функцию тождества).
- Термы **x** и **y** не альфа-эквивалентны, так как они не находятся в лямбда абстракции.

β-редукция

- Поскольку выражение **λ x . 2 ·x+1** обозначает функцию, ставящую в соответствие каждому **x** значение **2 ·x+1**, то для вычисления выражения **(λ x . 2 ·x+1) 3**, в которое входят и аппликация и абстракция, необходимо выполнить подстановку числа 3 в терм **2 ·x+1** вместо переменной **x**.
  В результате получается **2 ·3+1=7**.
- Это соображение в общем виде записывается как
  (λ x . t) a = t[x:=a]
  и носит название β-редукция.
- Выражение вида **(λ x.t) a**, то есть применение абстракции к некому терму, называется редексом (redex).
- β-редукция по сути является единственной «существенной» аксиомой λ-исчисления, она приводит к весьма содержательной и сложной теории. Вместе с ней λ-исчисление обладает свойством полноты по Тьюрингу и, следовательно, представляет собой простейший язык программирования.

η-преобразование

- η-преобразование выражает ту идею, что две функции являются идентичными тогда и только тогда, когда, будучи применёнными к любому аргументу, дают одинаковые результаты.
- η-преобразование переводит друг в друга формулы **λ x.f x** и **f.**
  (только если x не имеет свободных вхождений в f: иначе, свободная переменная x после преобразования станет связанной внешней абстракцией или наоборот).

Каррирование (карринг)

- Функция двух переменных **x** и **y**
  f (x,y) = x+y может быть рассмотрена как функция одной переменной **x**, возвращающая функцию одной переменной **y**, то есть как выражение
  λ x . λ y . x + y.
- На Python тоже самое

```
f=lambda x : lambda y : x+y
print(f(5)(3))
```

- Описанный процесс превращения функций многих переменных в функцию одной переменной называется **карринг** (также: каррирование), в честь американского математика Хаскелла Карри, хотя первым его предложил М.Э. Шейнфинкель (1924).

Чем это хорошо в языках?

- Повышение надёжности кода
- Удобство организации модульного тестирования
- Возможности оптимизации при компиляции
- Возможности параллелизма

Недостатки

- Отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти.
- Нестрогая модель вычислений приводит к непредсказуемому порядку вызова функций, что создает проблемы при вводе-выводе.
- Для преодоления недостатков функциональных программ уже первые языки функционального программирования включали механизмы императивного программирования.
- В чистых функциональных языках эти проблемы решаются другими средствами, например, в языке Haskell ввод-вывод реализован при помощи монад – нетривиальной концепции, позаимствованной из теории категорий.

Хотите знать больше?

- http://pv.bstu.ru/flp/fpLectures.pdf
- http://neerc.ifmo.ru/wiki/index.php?title=Лямбда-исчисление
- https://habrahabr.ru/post/215807/
- https://habrahabr.ru/post/215991/

Недоеденные плюшки

- Замена условных операторов
- **if** <условие>:

  <выражение 1>

  else:

  <выражение 2>
- **lambda**: (<условие> and <выражение 1>) or (<выражение 2>)

Недоеденные плюшки

- Функциональные циклы

- **for** e **in** lst:

  func(e)

- map(func,lst)

Недоеденные плюшки

- Последовательности действий

  do_it = **lambda** f : f()

  map(do_it, [f1, f2, f3])

Недоеденные плюшки

- Цикл while

```
while <cond>:
    <pre-suite>
    if <break_condition>:
        break
    else:
        <suite>
```

```
def while_block():
    <pre-suite>
    if <break_condition>:
        return 1
    else:
        <suite>
    return 0

while_FP =
lambda: (<cond> and while_block()) or while_FP()
while_FP()
```

Недоеденные плюшки

- Списковые включения, генераторы

```
>>> l = [x**2 for x in range(1,10)]


>>> numbers = [10, 4, 2, -1, 6]
>>> [x for x in numbers if x < 5]
[4, 2, -1]
```

Недоеденные плюшки

- Частичное применение функций или Карринг

```
def spam(x,y):
    print('param1={}, param2={}'.format(x,y))

spam1=lambda x : lambda y : spam(x,y)

def spam2( x ) :
    def new_spam( y ) :
        return spam( x, y )
    return new_spam

spam1(2)(3)       # карринг
spam2(2)(3)
s12=spam1(2)       # частичное применение
s12(4)
```

Вывод:

```
param1=2, param2=3
param1=2, param2=3
param1=2, param2=4
```

Недоеденные плюшки

- Итераторы

```
x=[1,2,3,4]
it=iter(x)
print(next(it))
print(it.__next__())
```

Эквиваленты

```
for i in iter(obj):
    print(i)
<=>
for i in obj:
    print(i)
```

Недоеденные плюшки

- Материализация итератора

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

Недоеденные плюшки

- Встроенные функции

  enumerate(iter) – нумерует итерируемый объект, возвращает словарь

```
print(dict(enumerate(['subject', 'verb', 'object'])))
{0: 'subject', 1: 'verb', 2: 'object'}
```

  sorted(iterable, key=None, reverse=False) – возвращает отсортированый список. sort() сортирует список на месте.

```
>>> l=[3,2,1,7]
>>> sorted(l)
[1, 2, 3, 7]
>>> l.sort()
>>> l
[1, 2, 3, 7]
```

reversed(iterable) – возвращает обращённый список.

Недоеденные плюшки

- Встроенные функции

  any(iter) и all(iter)

```
>>> any([0,1,0])
True
>>> any([0,0,0])
False
>>> any([1,1,1])
True
>>> all([0,1,0])
False
>>> all([0,0,0])
False
>>> all([1,1,1])
True
```

  zip(iterA, iterB, ...)

```
zip(['a', 'b', 'c'], (1, 2, 3)) =>
  ('a', 1), ('b', 2), ('c', 3)
```

Недоеденные плюшки

- Неограниченное количество параметров в лямбда функции:

```
>>> fun = lambda *args: args
>>> fun(5,4,10,7)
(5, 4, 10, 7)
```

Недоеденные плюшки

- Рекурсия. В функциональном программировании рекурсия является основным механизмом, аналогично циклам в итеративном программировании. Глубина ограничена 1000, но параметр может быть изменен.
- Примеры
- factorial = lambda x: ((x == 1) and 1) or x * factorial(x - 1)
- factorial = lambda z: reduce(lambda x, y: x * y, range(1, z + 1))

Исключение побочных эффектов

```
xs = (1, 2, 3, 4)
ys = (10, 15, 3, 22)
bigmuls = []
for x in xs:
    for y in ys:
        if x * y > 25:
            bigmuls.append((x, y))
print(bigmuls)
```

Исключение побочных эффектов

Зачем всё это?!

David Mertz:

«Огромный процент программных ошибок и главная проблема, требующая применения отладчиков, случается из-за того, что переменные получают неверные значения в процессе выполнения программы. Функциональное программирование обходит эту проблему, просто вовсе не присваивая значения переменным»

```
print([(x, y) for x in (1, 2, 3, 4) \
              for y in (10, 15, 3, 22) \
              if x * y > 25])
```

Задача

Что вычисляет алгоритм?

```
list1=[1, 2, 3]
list2=reversed(list1)

Resault=reduce(lambda res,x:res+x,\
map(lambda x,y:x+y, list1, list2))/2
print(resault)
```

Источники

- https://docs.python.org/dev/howto/functional.html
- https://ru.wikipedia.org/wiki/https://ru.wikipedia.org/wiki/Функциональное_программирование_на_Python
- Functional Programming in Python by David Mert

Вопросы