CSC326 Functional Programming

Jianwen Zhu <jzhu@eecg.toronto.edu>

Revision History

Revision 1.1	2017-10	JZ

Table of Contents

1. Agenda

- Overview
- From Imperative to Functional
- Closure
- Curry
- Continuation

2. Overview

- You have seen some elements of functional programming!
- Function as first class citizen
- Recursion used as primary control structure (no loop)
- Discourage or reject statements (use function application)
- Focus on list processing

- No side effect (no assignment, only name binding)
- High order: function that operates on functions that operates on functions
- Advanced constructs: Closure, Curry and Continuation

3. From Imperative to Functional

- Recall Dataflow machine: no storage, no state.
- Basic elements
  - map(func, list)
  - reduce(func, list)
  - filter(func, list)
  - lambda x : expr(x)

We now consider the methodology convert a regular imperative program into functional program. Note that our purpose is obtain insights on how to "express" in functional programming style. In no way we are advocating a process where you first program in imperative style, then converting to functional program. Once we obtain a solid understanding, we should always "think" directly in a functional way.

3.1. Eliminating if

- Imperative

```
# Normal statement-based flow control
if <cond1>:   func1()
elif <cond2>: func2()
else:         func3()
```

- Functional

```
    #------ "Short-circuit" conditional calls in Python -----#

    # Equivalent "short circuit" expression
    (<cond1> and func1()) or (<cond2> and func2()) or (func3())

    # Example "short circuit" expression
    >>> x = 3
    >>> def pr(s): return s
    >>> (x==1 and pr('one')) or (x==2 and pr('two')) or (pr('other'))
    'other'
    >>> x = 2
    >>> (x==1 and pr('one')) or (x==2 and pr('two')) or (pr('other'))
    'two'
```

[note] func1() etc have to return a non-zero value, otherwise result not as expected.

```
    #--------- Lambda with short-circuiting in Python -------#
    >>> pr = lambda s:s
    >>> namenum = lambda x: (x==1 and pr("one")) \
    ...                  or (x==2 and pr("two")) \
    ...                  or (pr("other"))
    >>> namenum(1)
    'one'
    >>> namenum(2)
    'two'
    >>> namenum(3)
    'other'
```

3.2. Eliminating Sequential Statement

- Imperative

```
#---------- Functional 'for' looping in Python ----------#
for e in lst:  func(e)      # statement-based loop
map(func,lst)               # map()-based loop
```

- Functional [source,python]

```
    #----- Functional sequential actions in Python ----------#
    # let's create an execution utility function
    do_it = lambda f: f()

    # let f1, f2, f3 (etc) be functions that perform actions
    map(do_it, [f1,f2,f3])    # map()-based action sequence
```

3.3. Eliminating While Statement

- Imperative

```
#-------- Functional 'while' looping in Python ----------#
# statement-based while loop
while <cond>:
    <pre-suite>
    if <break_condition>:
        break
    else:
        <suite>
```

- Functional

```
# FP-style recursive while loop
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()
```

- Imperative

```
# imperative version of "echo()"
```

```
def echo_IMP():
    while 1:
        x = raw_input("IMP -- ")
        if x == 'quit':
            break
        else:
            print x
echo_IMP()
```

- Functional

```
# utility function for "identity with side-effect"
def monadic_print(x):
    print x
    return x
# FP version of "echo()"
echo_FP = lambda: monadic_print(raw_input("FP -- "))=='quit' or echo_FP()
echo_FP()
```

3.4. Eliminating Side Effect

- Imperative

```
#--- Imperative Python code for "print big products" ----#
# Nested loop procedural style for finding big products
xs = (1,2,3,4)
ys = (10,15,3,22)
bigmuls = []
# ...more stuff...
for x in xs:
    for y in ys:
        # ...more stuff...
        if x*y > 25:
            bigmuls.append((x,y))
            # ...more stuff...
# ...more stuff...
print bigmuls
```

- Functional

```
#--- Functional Python code for "print big products" ----#
bigmuls = lambda xs,ys: filter(lambda (x,y):x*y > 25, combine(xs,ys))
combine = lambda xs,ys: map(None, xs*len(ys), dupelms(ys,len(xs)))
dupelms = lambda lst,n: reduce(lambda s,t:s+t, map(lambda l,n=n: [l]*n, lst))
print bigmuls((1,2,3,4),(10,15,3,22))
```

- A much better syntax (you knew it already!)

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

3.5. High Order Function

A function that meet at least one of the following criteria is called a higher-order function.

- takes one or more functions as arguments
- returns a function as its result

In fact, A Python function is not only a higher-order function, but also a first-class function, which satisfies following four criteria:

- can be created at runtime
- can be assigned to a variable
- can be passed as a argument to a function
- can be returned as the result of a function
- map, reduce, filter are high order functions
- Roll your own

```
>>> def add1(n) : return n+1
...
>>> def add2(n) : return n+2
...
>>> def compose(a,b) : return lambda n : a(b(n))
...
>>> add3 = compose(add1,add2)
>>>
>>> add3(1)
4
```

4. Closure

You have already seen the use of nested function in the context of meta programming using decorators. The returned nested function is in fact a closure, an important concept with many applications.

In a nutshell, a closure is a function bundled with data. In the context of conventional programming languages such as C, which does not have direct language level support for closure, it comes with the name of "callback" function. In Python, closure is naturally supported.

Read Codevoila: https://www.codevoila.com/post/51/python-tutorial-python-function-and-python-closure.

5. Currying

Being able to create closure on-the-fly allows us to implement an important concept in functional programming, called currying, which can be considered as incremental binding of functional parameters to specialize a general function.

Read Massimiliano Tomassoli: https://mtomassoli.wordpress.com/2012/03/18/currying-in-python

6. Continuation

Another important application of is to implement continuation, and the continuation passing style (CPS).

6.1. CPS in Javascript

We start by examples in Javascripts.

Read Matt Might: http://matt.might.net/articles/by-example-continuation-passing-style/

We now reproduce some of the examples in Python.

6.2. CPS in Python

- Factorial

```
# continuation-passing-style (CPS) for factorial
def cps_factorial( n, ret ) :
    if n == 0 :
        ret( 1 )
    else :
        cps_factorial( n-1, lambda t0 : ret (n*t0) )

def pr( n ) :
    print n

cps_factorial( 5, pr )
```

- Exceptional Handling

```
# continuation-passing-style (CPS) with exception handling
def cps_factorial_with_except( n, ret, err ) :
    if n < 0 :
        err( n )
    elif n == 0 :
        ret( 1 )
    else :
        cps_factorial_with_except( n-1, lambda t0:ret(n*t0), err )

def error( n ) :
    print "exception handled ", n

cps_factorial_with_except( 5, pr, error )
cps_factorial_with_except( -1, pr, error )
```

6.3. Probabilistic Programming Example with CPS

Consider a problem where we want to model an experiment where we flip a coin three times and we would like to find out the distribution of the outcome. In the term of mathematics, we are to find the distribution of a binomial process, given a bernoulli process with a bias of p. A distribution is typically represented by its probability mass function (PMF), or simply a histogram.

We can write a Python program to simulate the experiment, as follows. Note here the function bernoulli() returns the PMF of the bernoulli process (flipping a coin with a probability of p facing up); and sample() draws one sample from the given distribution.

```
# regular binomial
def bernoulli( p ) :
    return { 0:p, 1:1-p }

def sample( dist ) :
    return dist.keys()[0]

def binomial() :
    a = sample(bernoulli(0.5))
    b = sample(bernoulli(0.5))
    c = sample(bernoulli(0.5))
    return (a, b, c)

def explore_first( comp ) :
    return comp()

x = explore_first( binomial )
print x
```

The above code is intuitive to write, and runs fine, but the trouble is that it only gives one sample (0,0,0) of the binomial process. To find the complete histogram of the binomial process, we wish to enumerate all possible sample values. Examining closely, we find that the the sample() function returns, deterministicly, only 1 value (the first of the distribution), whereas ideally, it should have 2 possible outcome.

So in conventional programming, which is deterministic, every function call, at runtime, returns exactly 1 value. We now have the requirement that 1 or more function might return multiple different values, and we want to explore all such outcomes. In this specific example, there are three callsites of sample(bernoulli(0.5)), and each could return 0 or 1, and we would really want to execute ALL 8 possible scenarios.

In other words, in conventional programming, each run explore only execution path through the program. What if we want to exhaustively execute all possible execution paths of a program?

Now let’s work out the magic using CPS, as follows. Here we introduce a list unexploredFutures which captures a list of continuations, which are closures to be executed. In the new cps_sample(), which is a replacement of sample(), it enumerate all possible sample of a distribution (dist.keys()), create a corresponding continuation, and then append to unexploredFutures. One of the continuation is popped from the list and executed.

Once the first continuation is executed until the end, the exit() function is called, which will pop another continuation for execution. This way, all possible execution path is executed!

```
# cps binomial
unexploredFutures = []
def cps_sample( dist, cont ) :
    map( lambda s : unexploredFutures.append(lambda:cont(s)), dist.keys() )
    unexploredFutures.pop()()

def cps_binomial( ret ) :
    cps_sample( bernoulli(0.5), lambda a : \
    cps_sample( bernoulli(0.5), lambda b : \
    cps_sample( bernoulli(0.5), lambda c : \
                ret( (a,b,c) ) ) ) )

def exit( val ) :
    print val
    if len(unexploredFutures) > 0 :
        unexploredFutures.pop(0)()

def explore_cps( cpsComp ) :
    cpsComp( exit )

print "explore_cps -----------------------"
explore_cps( cps_binomial )
```

7. Recap

- Functional Programming is getting more popular
- Everything is an expression
  - Function application (with recursion)
  - Lambda
  - Short-circuit
- The 3Cs: Closure / Currying / Continuation

8. Disclaimer

The above lecture notes has used selected materials originally published by the following authors hereby acknowledged:

- David Mertz (https://www.ibm.com/developerworks/library/l-prog/index.html)
- Matt Might (http://matt.might.net/articles/by-example-continuation-passing-style)
- Massimiliano Tomassoli (https://mtomassoli.wordpress.com/2012/03/18/currying-in-python)
- Codevoila (https://www.codevoila.com/post/51/python-tutorial-python-function-and-python-closure)