Functional Programming

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

10, 2020

Outline

Function are values, i.e., a function can be

	Value	Function
• Anonymous	3	x => x + 1
• Assigned to a variable	x = 3	f = x => x + 1
• Passed as input/output parameter	f(3)	f(x => x + 1)
• Created dynamically	3 + 4	$f \circ g$

Fundamental Theory

- Imperative languages $\Rightarrow$ Von Neumann Architecture
  - Efficiency
- Functional languages $\Rightarrow$ Lambda Calculus
  - A solid theoretical basis that is also closer to the user, but
  - relatively unconcerned with the architecture of the machines on which programs will run

- A mathematical function is
  - a mapping of members of one set, called the domain set, to another set, called the range set
- A **lambda expression** specifies the parameter(s) and an expression in the following form to express an anonymous function
  λ(x) x * x * x
  like the function cube cube(x) = x * x * x
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression
  (λ(x) x * x * x)(2) which evaluates to 8

- A higher-order function is one that either takes functions as parameters or yields a function as its result, or both
- For example,
  - Function composition
  - Apply-to-all
  - Forall/Exists
  - Insert-left/Insert-right
  - Functions as parameters
  - Closures

Function Composition

A function that

- takes two functions as parameters and
- yields a function whose value is the first actual parameter function applied to the application of the second

  $f \circ g = f : (g : x)$

  For $f(x) = x + 2$; $g(x) = x * x$; $f \circ g\ (x) = x * x + 2$

Example in Scala,

```
val f = (x:Double) => x + 2
val g = (x:Double) => x * x
val h = f compose g
h(3)
val k = f andThen g
k(3)
```

Apply-to-all

A functional form that

- takes a single function as a parameter and
- yields a list of values obtained by applying the given function to each element of a list of parameters

$$\alpha f :< x_1, x_2, ..., x_n >=< f : x_1, f : x_2, ..., f : x_2 >$$

For h(x)=x*x $\Rightarrow$ αh:(1,2,3) yields (1,4,9)

Example in Scala,

```
List(2,3,4).map((x:Int) => x * x)
def inc (x:Int) = x + 1
List(4,5,6).map(inc)
```

A functional form that

- takes a single **predicate** function as a parameter and
- yields a value obtained by applying the given function to each element of a list of parameters and take the **and/or** of the results

$$\forall f :< x_1, x_2, ..., x_n >= \bigcap f : x_i$$
$$\exists f :< x_1, x_2, ..., x_n >= \bigcup f : x_i$$

Example in Scala,

```
def isEqualToThree(x:Int) = x == 3
List(2,3,4).forall(isEqualToThree)
               // yield false
List(2,3,4).exists(isEqualToThree)
               // yield true
```

Filter

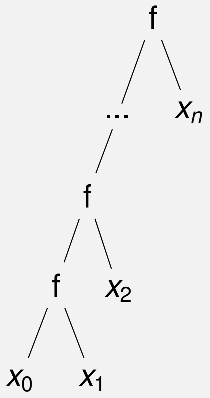
A functional form that

- takes a single **predicate** function as a parameter and
- yields a value that includes elements from the list parameter on which the given function returns true

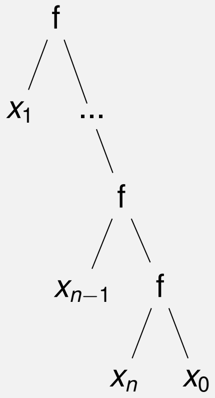
$$\beta f :< x_1, x_2, ..., x_n >=< x_i | f : x_i = T >$$

Insert Left / Insert Right

$/f :< x_0 >, < x_1, x_2, ..., x_n >$



$\backslash f :< x_0 >, < x_1, x_2, ..., x_n >$



Example in Scala

```
List(2,3,4).foldLeft(0)((a,b) => a+b) // yield 9
List(2,3,4).foldLeft(1)((a,b) => a*b) // yield 24
List(2,3,4).foldLeft("A")((a,b) => a + b)
                            // yield "A234"
List(2,3,4).foldRight("A")((a,b) => a + b)
                           // yield "234A"
```

In user-defined functions, functions can be passed as parameters.

```
def apply(x:Int)(f:Int=>Int) = f(x)
val inc1 = (x:Int) => x + 1
val sq = (x:Int) => x * x
val fl = List(inc1,sq)
fl.map(apply(3)) //yield List(4,9)
```

Closure [1]

"An object is data with functions. A closure is a function with data." - John D. Cook

```
def power(exp:Double) =
          (x:Double) => math.pow(x,exp)
val square = power(2)
square(4) //yield 16.0
val cube = power(3)
cube(3) //yield 27.0
```

Closure = function + binding of its free variables

Curying functions [2]

$f : X_1 \times X_2 \times ... \times X_n \to Y$

curry: $f : X_1 \to X_2 \to ... \to X_n \to Y$

Example in Scala

```
def add(x:Int, y:Int) = x + y
add(1,3)
add(1) add(1)(3)
def plus(x:Int)(y:Int) = x + y
plus(1)(3)
val inc1 = plus(1) _
inc1(3)
val addCurried = (add _).curried
val plusUncurried = Function.uncurried(plus _)
```

Read more on Partially Applied Functions [2]

Immutability

- Immutable: Cannot change
- In Java, strings are immutable
  "Hello".toUpper() doesn't change "Hello" but returns a new string "HELLO"
- In Scala, **val** is immutable
  ```
  val num = 12
  num = 10 // wrong
  ```
- Pure functional programming: No mutations
- Don't mutate—always return the result as a new value
- Functions that don't mutate state are inherently parallelizable

Example on Immutability

```
abstract class IntStack
   def push(x: Int): IntStack =
                  new IntNonEmptyStack(x, this)
   def isEmpty: Boolean
   def top: Int
   def pop: IntStack
class StackEmpty extends IntStack
   def isEmpty = true
   def top = error("EmptyStack.top")
   def pop = error("EmptyStack.pop")
class IntNonEmptyStack(elem: Int, rest: IntStack)
                              extends IntStack
   def isEmpty = false
   def top = elem
   def pop = rest
```

- Immutable Data Structures
- lambda function
- First-class functions
- High-order functions: map, filter, reduce
- Closure
- Decorator

This is a style of programming that avoids side effects by performing computation

- through the evaluation of pure functions
- relying heavily on immutable data structures

Benefits of functional programming:

- Pure functions are easier to reason about
- Testing is easier
- Debugging is easier
- Programs are more bulletproof
- Parallel/Concurrent programming is easier

Immutable Data Structures

- Apply immutable data types in Python
  - Number (int,float,complex)
  - Boolean (bool)
  - String (str)
  - Sequence (tuple, range)
    (1,'a',True)
  - Set (frozenset)
    x = frozenset({1,'a',True})
  - Mapping (collections.namedtuple)
    Student = collections.namedtuple('Student',['name','age'])
    x = Student('Vinh',14)
    x.name
- Apply immutable operations on mutable data types
  - + instead of **extend()**
    [1,2,3] + [4,5]

Lambda function

- Syntax:

```
lambda (<param>(, <param>)*)?: <exp>
```

- For example,

```
lambda a,b:  a + b
(lambda a,b:  a + b)(3,4) => 7
x = lambda a,b:  a + b
x(3,4) => 7
```

- Anonymous function
- Any number of parameters
- Body is just one expression
- Used in high-order functions

First-class function

- A function is treated as any other value, i.e. it is
  - assigned to a variable

```
def foo(a,b): pass
x = foo
x(3,4)
```

  - passed into another function as a parameter

```
def foo(f,x):
  return f(x)
foo(lambda a: a ** 2, 4) => 16
```

  - returned as a value

```
def f(x):
  def g(y):
    return x * y
  return g
m = f(3)
m(4) => 12
```

High-order functions: map, filter, reduce

- **map(<function>,<sequence>)**: apply **<function>** to each element of **<sequence>** and return an iterator

```
cels = [36.5, 37, 37.5, 38, 39]
fahr = list(map(lambda c:  (float(9) / 5) * c
+ 32,cels))
=> [97.7, 98.6, 99.5, 100.4, 102.2]list(map(lamb
x,y:  x + y,[1,2,3],[4,5,6,7])) => [5,7,9]
```

- **filter(<function>,<sequence>)** return an iterator that contains elements in <sequence> for which <function> returns True

```
list(filter(lambda c:  c % 2 == 1, [0,1,2,3,4,5]
[1,3,5]
```

- **reduce(<function>,<sequence>(,<initial>)?)**: if <sequence> is [s_1,s_2,s_3], **reduce** return function(function(s_1,s_2),s_3) or function(function(function(<initial>,s_1),s_2),s_3)

```
from functools import reduce
```

- **Closure** is a function object together with an environment (binding of its free data).

```
def power(y):
  def inner(x):
    return x ** y
  return inner
square= power(2)
square(5) => 25
```

- **Decorator** allows to modify the behavior of function or class without permanently modifying it.

```
@log_decorator
def foo(x,y):
  return x*y
print(foo(3,4)) => 12
    => foo is running
    => 12
```

```
def foo(x,y):
  return x*y
foo = log_decorator(foo)
print(foo(3,4))
```

- How?

```
def log_decorator(func):
  def inner(*arg):
    print(func.__name__+" is running")
    return func(*arg)
  return inner
```

Summary

- Functional programming languages use **function application**, **conditional expressions**, **recursion**, and **functional forms** to control program execution instead of imperative features such as variables and assignments
- Purely functional languages have advantages over imperative alternatives, but their **lower efficiency** on existing machine architectures has prevented them from enjoying widespread use

[1] Methods and Closures, http://www.artima.com/pins1ed/functions-and-closures.html, 19 06 2014.

[2] Function Currying in Scala, http://www.codecommit.com/blog/scala/function-currying-in-scala, 19 06 2014.

[3] Case classes and pattern matching, http://www.artima.com/pins1ed/case-classes-and-pattern-matching.html, 19 06 2014.

[4] Control Abstraction, http://www.artima.com/pins1ed/control-abstraction.html, 19 06 2014.