First-Class Functions

October 4, 2016

Riccardo Pucella

What we've done till now

We've developed a core interpreter

- parser for surface syntax
- representation
- evaluator for execution

Object language: a simple expression language

- a glorified calculator

What’s next?

We take our core interpreter, and extend it in various directions to capture existing programming models

Next up: we beef up expressions
— leads to functional programming languages

(Later: instead of beefing up expressions, we add statements that can modify the state)

Functional programming

Functional programming languages are characterized by:

— everything is an expression returning a value
— functions are first-class citizens
  – they can be created and passed around like any other value without any restriction

In its purest form:

— evaluation has no side-effects
— evaluation is lazy (call-by-name)

Higher-order functions

A higher-order function is a function that takes another function as argument

```
def succ (x):
  return x + 1

def twice (f,x):
  return f(f(x))

twice(succ,10) →
succ(succ(10)) →
12
```

Utility of higher-order functions

Higher-order functions can capture the structure of a family of functions

```
# return a list with all elements doubled
def doubles (lst):
  result = []
  for elem in lst:
    result.append(2*elem)
  return result
```

Utility of higher-order functions

Higher-order functions can capture the structure of a family of functions

```
# return a list with all last digits of elements
def last_digits (lst):
  result = []
  for elem in lst:
    result.append(elem % 10)
  return result
```

Utility of higher-order functions

Higher-order functions can capture the structure of a family of functions

```
# return a list with all elements transformed via f
def map (lst,f):
  result = []
  for elem in lst:
    result.append(f(elem))
  return result
```

Utility of higher-order functions

Higher-order functions can capture the structure of a family of functions

```
# return a list with all elements transformed via f
def map (lst,f):
  result = []
  for elem in lst:
    result.append(f(elem))
  return result

def doubles (lst):
  def double (x):
    return 2*x
  return map(lst,double)
```

Utility of higher-order functions

Higher-order functions can capture the structure of a family of functions

```
# return a list with all elements transformed via f
def map (lst,f):
  result = []
  for elem in lst:
    result.append(f(elem))
  return result

def last_digits (lst):
  def last_digit (x):
    return x % 10
  return map(lst,last_digits)
```

Another example: filtering

```
def evens (lst):
   result = []
   for elem in lst:
     if elem % 2 == 0:
        result.append(elem)
   return result
```

Another example: filtering

```
def evens (lst):
   result = []
   for elem in lst:
     if is_even(elem):
        result.append(elem)
   return result

def is_even (x):
  return x % 2 == 0
```

Another example: filtering

```
def filter (lst,p):
   result = []
   for elem in lst:
     if p(elem):
        result.append(elem)
   return result

def is_even (x):
  return x % 2 == 0

def evens (lst):
  return filter(lst,is_even)
```

Another example: reducing

```
def sum (lst):
  result = 0
  for elem in lst:
     result += elem
  return result
```

Another example: reducing

```
def sum (lst):
  result = 0
  for elem in lst:
     result = add(result,elem)
  return result

def add (x,y):
  return x + y
```

Another example: reducing

```
def reduce (lst,init,f):
  result = init
  for elem in lst:
     result = f(result,elem)
  return result

def add (x,y):
  return x + y

def sum (lst):
  return reduce(lst,0,add)
```

Another example: reducing

```
def reduce (lst,init,f):
  result = init
  for elem in lst:
     result = f(result,elem)
  return result

def append (x,y):   # append two lists
  return x + y

def flatten (lst):
  return reduce(lst,[],append)
```

Anonymous functions

Giving a name to a function just to pass it to another function is a pain

Sometimes, you just want a function without needing it to have a name

```
def double (x):
   return 2 * x

def doubles (lst):
   return map(lst,double)
```

Anonymous functions

Giving a name to a function just to pass it to another function is a pain

Sometimes, you just want a function without needing it to have a name

```
 ~~def double (x):~~
 ~~   return 2 * x~~

 def doubles (lst):
    return map(lst,lambda x: 2*x)
```

Functions returning functions

Functions can be returned as values:

```
def double (x):
   return 2 * x

def triple (x):
   return 3 * x

…
```

Functions returning functions

Functions can be returned as values:

```
def ktimes (k):
  def mult_by_k (x):
    return k*x
  return mult_by_k

double = ktimes(2)

triple = ktimes(3)
```

Functions returning functions

Functions can be returned as values:

```
def ktimes (k):
  return lambda x : k * x




double = ktimes(2)

triple = ktimes(3)
```

Example: composing functions

```
def compose (f,g):
  return lambda x : g(f(x))

def triple (x):
  return 3 * x
def add1 (x):
  return x + 1

(compose(triple,add1))(10)
(compose(add1,triple))(10)
```

A functional object language

We want to extend our expression language with first-class functions

Features:

- New value representing a function
- `ECall` taking an expression as first argument
- Using the environment model
  - lets us merge the function dictionary into the environment

Functions are values

We want to be able to write:

```
(defun add1 (x) (+ x 1))
(defun identify (f) f)

(identity add1)
```

Note that `(identity add1)` evaluates to the function `add1`

Since expressions evaluate to values, we need a value representing functions

ECall requires an expression

```
(defun add1 (x) (+ x 1))
(defun twice (f)
   (function (x) (f (f x))))
```

We should be able to write

```
  (let ((f (twice add1)))
     (f 10))
```

But that means we should also be able to write

```
  ((twice add1) 10)
```

Environment model

Evaluate an identifier by looking it up in the environment

We can turn the function dictionary into an environment, associating names of functions to the function value that defines them.

Function dictionary = INITIAL ENVIRONMENT

Evaluation is simple: .eval(*env)*

A functional object language

```
EValue (value)      EInteger(i), EBoolean(b)
EPrimCall (symbol,exps)
EIf (exp,exp,exp)
ELet (bindings, exp)
EId (symbol)
ECall (exp,exp)
EFunction (parameter,exp)

VInteger (i)
VBoolean (b)
VFunction (parameter,exp)
```

A functional object language

```
EValue (value)     EInteger(i), EBoolean(b)
EPrimCall (symbol,exps)
EIf (exp,exp,exp)
~~ELet (bindings, exp)~~
EId (symbol)
ECall (exp,exp)
EFunction (parameter,exp)

VInteger (i)
VBoolean (b)
VFunction (parameter,exp)
```

Surface syntax

```
atomic ::= integer
           symbol
           true
           false

expr ::= atomic
         ( if expr expr expr )
         ( let ( ( symbol expr ) … ) expr )
         ( function ( symbol ) expr )
         ( expr expr )
```

Implementation (take 1)

```
class VFunction (Value):
  def __init__ (self,param,body):
    self.param = param
    self.body = body
    self.type = "function"


class EFunction (Value):
  def __init__ (self,param,body):
    self._param = param
    self._body = body

  def eval (self,env):
    return VFunction(self._param,self._body)
```

Implementation (take 1)

```
class ECall (Exp):
  def __init__ (self,fun,arg):
    self._fun = fun
    self._arg = arg

  def eval (self,env):
    f = self._fun.eval(env)
    if f.type != "function":
      error
    arg = self._arg.eval(env)
    new_env = [(f.param,arg)] + env
    return f.body.eval(new_env)
```

Implementation (take 1)

```
class ECall (Exp):
  def __init__ (self,fun,arg):
    self._fun = fun
    self._arg = arg

  def eval (self,env):
    f = self._fun.eval(env)
    if f.type != "function":
      error
    arg = self._arg.eval(env)
    new_env = [(f.param,arg)] + env
    return f.body.eval(new_env)
```

What should this evaluate to?

```
(let ((x 10))
  (let ((f (function (y) ( + x y))))
    (f 100)))
```

What should this evaluate to?

```
(let ((x 10))
  (let ((f (function (y) ( + x y))))
    (f 100)))
```

```
(let ((x 10))
  (let ((f (function (y) (+ x y))))
    (let ((x 20))
      (f 100))))
```

Problem

This should return 110, according to the substitution model -- this is static binding

If it returns 120, you've implemented dynamic binding

(dynamic binding was popular in the 60s)

```
(le
  (
```

```
(let ((x 10))
  (let ((f (function (y) (+ x y))))
    (let ((x 20))
      (f 100))))
```

Binding strategies

A function may refer to identifiers that are not arguments to the function.
— where do we look up their value?

Dynamic binding: look for the value in the nearest enclosing bindings where the function is called

Static binding: look for the value in the nearest enclosing bindings where the function is defined

Closures

The problem of implementing static binding in the context of first-class functions is often called the *upwards FUNARG problem*

Solution: *record the environment that was present when a function was defined*

A function value that records the environment in which it was defined is called a closure

Implementation (take 2)

```
class VClosure (Value):
  def __init__ (self,param,body,env):
    self.param = param
    self.body = body
    self.env = env
    self.type = "function"

class EFunction (Value):
  def __init__ (self,param,body):
    self._param = param
    self._body = body

  def eval (self,env):
    return VClosure(self._param,self._body,env)
```

Implementation (take 2)

```
class ECall (Exp):
  def __init__ (self,fun,arg):
    self._fun = fun
    self._arg = arg

  def eval (self,env):
    f = self._fun.eval(env)
    if f.type != "function":
      error
    arg = self._arg.eval(env)
    new_env = [(f.param,arg)] + f.env
    return f.body.eval(new_env)
```