Functions

February 15, 2018

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 objects

  – 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)

Top level functions

Baby steps: let's add top level functions

Need:

- store of function definitions
- a way to apply (call) functions to values

We do it with an eye towards generality:

- we use the environment to store functions
- we use identifiers to refer to functions
- we have an EApply node

Function values (top-level)

```
class VFunction1 (val param: String, val body:Exp) extends Value {

  override def toString () : String =
      "<" + param + " -> " + body + ">"

  override def isFunction () : Boolean = true

  // what can you do with a function value?
  // you can apply it to a value

  override def apply1 (arg: Value) : Value = {
     return body.eval(new Env(List((param,arg))))
  }
}
```

Expression EApply1

```
class EApply1 (val f : Exp, val arg : Exp) extends Exp {

   override def toString () : String =
      "EApply1(" + f + "," + arg + ")"

   def eval (env : Env) : Value = {
      val vf = f.eval(env)
      val varg = arg.eval(env)
      return vf.apply1(varg)
   }
}
```

Demo functions-01.scala

(top-level functions)

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)
```

Demo functions-02.scala

(Higher-order functions, more or less)

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)
```

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

(sometimes called dynamic/static scoping)

Introduce some surface syntax

```
atomic ::= integer
           symbol
           true
           false

expr ::= atomic
         ( if expr expr expr )
         ( let ( ( symbol expr ) … ) expr )
         ( function1 ( symbol ) expr )
         ( expr expr )
```

What should this evaluate to?

```
(let ((x 10))
  (let ((f (function1 (y) ( + x y))))
    (f 100)))
```

What should this evaluate to?

```
(let ((x 10))
  (let ((f (function1 (y) ( + x y))))
    (f 100)))
```

```
(let ((x 10))
  (let ((f (function1 (y) (+ x y))))
    (let ((x 9000))
      (f 100))))
```

What should this evaluate to?

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

If it returns 9100, you've implemented dynamic binding

(dynamic binding was popular in the 60s because it was easier to implement)

```
(let ((x 10))
  (let ((f (function1 (y) (+ x y))))
    (let ((x 9000))
      (f 100))))
```

Demo functions-03.scala

(Dynamic binding implementation)

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

Demo functions-04.scala

(Static binding implementation)