Functions

February 6, 2020

Riccardo Pucella

Until now

- Simple expression language
  - Abstract representation as a tree of expressions
  - Evaluation: reduce an expression to a value
- Expressions with different types of values
- Different operations on those values
- Identifiers and bindings to values

Today

We add functions

- A function is a parameterized expression
- Use a function by applying it to arguments

Three approaches:

- Top-level function definitions
- Functions as values
- Functions as values and local definitions

(1) Functions definitions

First attempt

- Top level functions
- Available in a store of function definitions

Function application:

- Expression to apply a function (a name) to arguments (expressions)

Functions of one argument for now

- Generalized on the homework

Function environment

```
class FEnv (val content: List[(String, FunctionDef)]) {
  ...

  def lookup (id : String) : FunctionDef = {
    for (entry <- content) {
      if (entry._1 == id) {
        return entry._2
      }
    }
    throw new Exception("Unbound function identifier " + id)
  }
}
```

Revised expression evaluation

```
abstract class Exp {

  def eval (env : Env, fenv : FEnv) : Value

}
```

Function definition

```
class FunctionDef (val param : String, val body : Exp) {
  …

  def apply (arg : Value, fenv : FEnv) : Value =
    body.eval(new Env(List((param, arg))), fenv)
}
```

Example: squaring function

```
new FunctionDef ("x", new ETimes(new EId("x", new EId("x")))
```

Expression EApply

```
class EApply (val fn : String, val arg : Exp) extends Exp {
  ...

  def eval (env : Env, fenv : FEnv) : Value = {
    val varg = arg.eval(env, fenv)
    val df = fenv.lookup(fn)
    return df.apply(varg, fenv)
  }
}
```

Demo functions 1

(2) Function values

We now have two environments around:

- an environment for values
- an environment for functions

Might be simpler to use just one environment

- Make functions a kind a value

Nice side effect:

- We can pass functions as arguments
- We can store functions in data structures

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)
```

Extending Value class

```
abstract class Value {
  ...

  def isFunction () : Boolean = false

  def apply1 (v1 : Value, env : Env) : Value = {
    throw new Exception("Value not of type FUNCTION-1")
  }

  def apply2 (v1 : Value, v2 : Value, env : Env) : Value = {
    throw new Exception("Value not of type FUNCTION-2")
  }
}
```

Function value

```
class VFunction1 (val param1 : String, val body : Exp)
                                               extends Value {
  ...

  override def apply1 (arg1 : Value, env : Env) : Value = {
    val new_env = env.push(param1, arg1)
    return body.eval(new_env)
  }
}
```

Class `VFunction2` similar but takes 2 parameters

Expression EApply1

```
class EApply1 (val fn : String, val arg1 : Exp) extends Exp {
  ...

  def eval (env : Env) : Value = {
    val varg1 = arg1.eval(env)
    val vfn = env.lookup(fn)
    return vfn.apply1(varg1, env)
  }
}
```

Class `EApply2` similar but takes 2 arguments

Demo functions 2

Local function definitions

Right now, we have assumed that function values simply exist in the environment

- Presumably added to an initial environment

We have no way to "create" functions

- Need an expression that creates a function

Something like this in some toy language:

letfun (*f* (*x*) = *e*) *body*

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)

What should this evaluate to?

```
let (x = 10)
  letfun (f (y) = x + y)
    f 100
```

```
let (x = 10)
  letfun (f (y) = x + y)
    let (x = 9000)
      f 100
```

What should this evaluate to?

Static binding - returns 110

Dynamic binding - returns 9100

Dynamic binding was popular in the 60s
because it was easier to implement

```
let
  l
```

```
let (x = 10)
  letfun (f (y) = x + y)
    let (x = 9000)
      f 100
```

Expression ELetFun1

```
class ELetFun1 (val fn : String, val param1 : String,
                val fbody : Exp, val ebody : Exp) extends Exp {
  ...

  def eval (env : Env) : Value = {
    val vfn = new VFunction1(param1, fbody)
    val new_env = env.push(fn, vfn)
    return ebody.eval(new_env)
  }
}
```

Demo functions 3

Closures

How do we implement static binding?

Record the environment that was present when a function was defined with the function

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

(Historically called the *upwards FUNARG problem*)

Value VClosure1

```
class VClosure1 (val param1 : String, val body : Exp,
                 val env : Env) extends Value {
  ...

  override def apply1 (arg1 : Value) : Value = {
    val new_env = env.push(param1, arg1)
    return body.eval(new_env)
  }
}
```

This replaces `VFunction1`

Expression EApply1

```
class EApply1 (val fn : String, val arg1 : Exp) extends Exp {
  ...

  def eval (env : Env) : Value = {
    val varg1 = arg1.eval(env)
    val vfn = env.lookup(fn)
    return vfn.apply1(varg1)
  }
}
```

Expression ELetFun

```
class ELetFun1 (val fn : String, val param1 : String,
                val fbody : Exp, val ebody : Exp) extends Exp {
  ...

  def eval (env : Env) : Value = {
    val vfn = new VClosure1(param1, fbody, env)
    val new_env = env.push(fn, vfn)
    return ebody.eval(new_env)
  }
}
```

Demo functions 4

Homework

- Multiargument functions
- Anonymous functions
- Recursive functions!