Programming with Closures for Fun and Profit

G Bordyugov

ITB Meeting on 11.10.2016

What Closures Are

Nothing fancy: Just functions with captured state

```python
def makeAdder(y):
  def f(x):
    return x+y
  return f

add5  = makeAdder(5)
add13 = makeAdder(13)

add5(2)    # => 7
add13(13) # => 26
```

Pervasive: R, Python, JavaScript, ..., Clojure

Fifty years old, originating in APL and Lisp

Sharing and Hiding State

```python
def makeAccount(amount):
  money = [amount]
  def withdraw(x):
    money[0] -= x
    return money[0]
  def deposit(x):
    money[0] += x
    return money[0]

  return withdraw, deposit

withdrawA, depositA = makeAccount(100)
withdrawB, depositB = makeAccount(300)

withdrawA(10)  # => 90
withdrawB(100) # => 200
depositB(150)  # => 350
```

Building ODE Models

Solving numerically $x' = f(x, t, a, b, c, d, \ldots)$:

Having lots of parameters in the ODE often leads to

```
def rhs(x, t, a, b, c, d, ...)
  # calculate f of x, t, a, b, c, d, ...
  return f


# x0, t = ...
result = odeint(rhs,x0,t,a,b,c,d, ...)

# look at a particular point
rhs(x1, t1, a, b, c, d, ...)
```

Building ODE Models with Closures

```
def makeModel(a, b, c, d, ...):
  def rhs(x, t):
    # calculate f of x, t, a, b, c, d, ...
    return f
  return rhs

rhs1 = makeModel(0.1, 0.2, 0.3, 0.4, ...)
rhs2 = makeModel(0.4, 0.3, 0.2, 0.1, ...)

# x0, t = ...
result1 = odeint(rhs1, x0, t)
result2 = odeint(rhs2, x0, t)

# look at particular rhs's
rhs2(x1, t1)
```

Another Numerical Example

Automatic numerical differentiator

```
def makeDerivative(f, h=0.001):
  def derivative(x):
    return (f(x+h/2.0) - f(x-h/2.0))/h
  return derivative

dsin = makeDerivative(sin);
dsin(pi/2.0) # => 0.0

myDer = makeDerivative(myBigFunction)
# ...
```

- Here, we capture rather a *function* (the one to be differentiated) than a *state*

Memoizing/Caching Functions

- Case: function *f(x)* takes long time to compute, but happens to be called many times with a small number of *different* x
- Solution: to *memoize* (to *cache*) the results of *f(x)*
- Can be done on the fly using closures

Memoizing/Caching Functions

```python
def memoize(f):
  cache = {}
  def g(x):
    if not x in cache:
      cache[x] = f(x)
    return cache[x]
  return g

# f(x) is a "heavy" function
fmemoed = memoize(f)

f(x); f(x) # takes 2x time of f(x)

# the second call is for free
fmemoed(x); fmemoed(x)
```

Concatenating Lists

- Problem: list concatenation can be expensive if the first list is long: In order to do the concatenation `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] + [1, 2]`, we must go through all elements of the first list
- Gets worse if we have many concatenations all over the place, the associativity becomes important:

```
[0, 1, 2, 3] + ([4, 5, 6]+[7, 8, 9])
```

or

```
([0, 1, 2, 3]+[4, 5, 6]) + [7, 8, 9]
```

- How to ensure the right (as opposed to left) associativity?
- Solution: Difference Lists

Difference Lists

A list is represented by a function that prepends it to a given list

```
def dlist(x):
  def f(y):
    print("concing", x, "+", y)
    return x + y
  return f

def show(x):
  return x([])
```

Concatenation becomes a simple function composition:

```
def concat(x, y):
  def f(z):
    return x(y(z))
  return f
```

Difference Lists

```
one = dlist([0, 1, 2])
two = dlist([3, 4, 5])

onetwo = concat(one, two)
otot   = concat(onetwo, onetwo)

show(otot)

> concing [3, 4, 5] + []
> concing [0, 1, 2] + [3, 4, 5]
> concing [3, 4, 5] + [0, 1, 2, 3, 4, 5]
> concing [0, 1, 2] + [3, 4, 5, 0, 1, 2, 3, 4, 5]
```

Why does it work?