Linguagens de Programação

Fabio Mascarenhas - 2015.2

http://www.dcc.ufrj.br/~fabiom/lp

Objetos sem classes

- Um *objeto* tem duas visões: a de fora e a de dentro
- Visto de fora, um objeto é uma entidade opaca, para a qual podemos mandar *mensagens*; uma mensagem pode ter *argumentos*, que são outros objetos, e gera uma *resposta*, que também é um objeto
- Visto de dentro, um objeto tem um conjunto de *campos*, e um conjunto de métodos, que correspondem às mensagens que esses objetos podem responder
- Somente o código de um método tem acesso aos campos do objeto

Proto

- *proto* é em essência uma linguagem imperativa, como MicroC
- Só que os valores de proto agora podem ser *números* ou *objetos*
  - Objetos têm campos, que são endereços de memória
- Temos as mesmas operações de MicroC para números, mas não temos mais ponteiros
- A operação @ acessa um campo do objeto corrente

```
fun counter(n)
  object (n)
    def inc(n)
      @0 := @0 + n
    end
    def dec(n)
      @0 := @0 - n
    end
  end
end

let c = counter(0) in
  print(c.inc(4));
  print(c.dec(2))
end
```

self

- A função `eval` para as expressões de *proto* precisa receber mais um parâmetro: o *objeto corrente*
- São os campos desse objeto que o operador `@` manipula; quando enviamos uma mensagem a outro objeto, o corpo do método é avaliado com o outro objeto como objeto corrente
- Vamos chamar o objeto corrente de `self`, e expor ele para o programa com uma primitiva com esse nome

```
fun counter(n)
  object (n)
    def inc(n)
      @0 := @0 + n
    end
    def dec(n)
      self.inc(-n)
    end
  end
end

let c = counter(0) in
  print c.inc(4);
  print c.dec(2)
end
```

Identidade

- Cada avaliação de `object` produz um objeto novo
- A função counter é como um construtor de objetos

```
fun counter(n)
  object (n)
    def inc(n)
      @0 := @0 + n
    end
    def dec(n)
      self.inc(-n)
    end
  end
end

let c1 = counter(0),
    c2 = counter(0) in
  print c1.inc(5);
  print c2.inc(2);
  print c1.dec(3);
  print c1 == c2
end
```

Delegação

- Um objeto pode *delegar* a implementação de seus métodos para outro objeto
- Com isso temos uma espécie de *herança* da implementação do objeto para qual ele delega

```
fun counter(n)
  object (n)
    def inc(n)
      @0 := @0 + n
    end
    def dec(n)
      self.inc(-n)
    end
  end
end
fun dcounter(o)
  object (o)
    def inc(n)
      @0.inc(n)
    end
  end
end
let c1 = counter(0),
    c2 = dcounter(c1) in
  print c1.inc(5);
  print c2.inc(2)
end
```

Recursão aberta

- Delegação permite reutilizar a implementação dos métodos do objeto `counter` na implementação dos métodos de `dcounter`
- Mas só com delegação não temos *recursão aberta*
- O programa ao lado imprime -3: a implementação de `dec` em `dcounter` delega para a implementação de `dec` em `counter`, que chama `self.inc`, só que `self` é o objeto `counter` que está em `@0` de `dcounter`
- Na recurão aberta, `self` seria um objeto `dcounter`, e o programa imprimiria -6

```
fun counter(n)
  object (n)
    def inc(n)
      @0 := @0 + n
    end
    def dec(n)
      self.inc(-n)
    end
  end
end
fun dcounter(o)
  object (o)
    def inc(n)
      @0.inc(n*2)
    end
    def dec(n)
      @0.dec(n)
    end
  end
end
let c1 = counter(0),
    c2 = dcounter(c1) in
  print c2.dec(3)
end
```

Herança

- Para ter recursão aberta vamos introduzir uma forma implícita de delegação, a *herança de implementação*
- Um objeto vai poder extender outro objeto, o seu *protótipo*; ele ganha o mesmo número de campos do protótipo, e pode ter campos adicionais
- Se não achamos um método no objeto então continuamos a busca no seu protótipo
- Uma vez encontrado o método a chamada é feita normalmente

```
fun counter(n)
  object (n)
    def inc(n)
      @0 := @0 + n
    end
    def dec(n)
      self.inc(-n)
    end
  end
end
fun hcounter(o)
  object () extends o
    def inc(n)
      @0 := @0 + n * 2
    end
  end
end
let c1 = counter(0),
    c2 = hcounter(c1) in
  print c1.inc(2);
  print c2.dec(3)
end
```

`super`

- A chamada de um método tem duas partes: buscar o método e a chamada em si
- Se começarmos a busca no protótipo do objeto, mas fizermos a chamada com `self` sendo o próprio objeto, temos o comportamento de `super` nas linguagens OO
- `super` delega a implementação para o protótipo, mas mantém a recursão aberta; o programa ao lado imprime -6
- SELF e JavaScript implementam modelos OO parecidos com os de *proto*

```
fun counter(n)
  object (n)
    def inc(n)
      @0 := @0 + n
    end
    def dec(n)
      self.inc(-n)
    end
  end
end
fun hcounter(o)
  object () extends o
    def inc(n)
      super.inc(n*2)
    end
  end
end
let c1 = counter(0),
    c2 = hcounter(c1) in
  print c1.inc(2);
  print c2.dec(3) -- -6
end
```

Objetos são de valores alta ordem

- Um objeto carrega a implementação de seus métodos consigo
- Programar com objetos se parece mais com a programação usando funções de primeira classe do que a programação imperativa tradicional, que usa apenas funções de primeira ordem
- Funções de alta ordem, tipos algébricos, casamento de padrões, tudo isso pode ser simulado em *proto* usando objetos sem nem mesmo usar herança

```
fun vazia()
  object ()
    def imprime() 0 end
    def map(f) self end
  end
end

fun cons(h, t)
  object (h, t)
    def imprime()
      print(@0); @1.imprime()
    end
    def map(f)
      cons(f.apply(@0), @1.map(f))
    end
  end
end

let l1 = cons(1, cons(2, vazia())),
    l2 = l1.map(object ()
                  def apply(o) o * o end
                end) in
  l1.imprime(); l2.imprime()
end
```

Recursão aberta, de novo

- Herança e recursão aberta dão mais expressividade
- No programa ao lado, o a função tracer constrói uma versão da “função” f que imprime seu argumento
- Com a recursão aberta, mesmo chamadas recursivas têm seus argumentos impressos

```
fun tracer(f)
  object () extends f
    def apply(x)
      print(x);
      super.apply(x)
    end
  end
end

let fat = object ()
            def apply(n)
              if n < 2 then
                1
              else
                n * self.apply(
                        n - 1)
              end
            end
          end in
  print(fat.apply(5));
  tracer(fat).apply(5)
end
```

Classes

- Uma classe é um molde para construir objetos
- A linguagem *proto* não tem classes na sua sintaxe, mas elas estão lá implicitamente, no número de campos do objeto e nos seus métodos
- Na maioria das linguagens OO o conceito de classe é bem mais explícito
- Uma classe por si só apenas descreve objetos; para instanciá-los usa-se uma operação primitiva de instanciação

Listas usando classes

- A primitiva `new` precisa do nome da classe que ela tem que instanciar, e dos valores para os campos
- As classes não são valores, a única coisa que podemos fazer com uma classe é instanciá-la
- Em essência, esse é o modelo OO de Java; os campos e métodos estáticos são simplesmente variáveis globais e funções com regras de escopo específicas

```
class vazia(0)
  def imprime() 0 end
  def map(f) self end
end

class cons(2)
  def imprime()
    print(@0); (@1).imprime()
  end
  def map(f)
    new cons(f.apply(@0),
             (@1).map(f))
  end
end

class quadrado(0)
  def apply(o) o * o end
end

let l1 = new cons(1,
             new cons(2,
                  new vazia())),
    l2 = l1.map(new quadrado()) in
  l1.imprime(); l2.imprime()
end
```

Classes de primeira classe

- Em linguagens como Smalltalk e Ruby, classes também são objetos, que podem ter seus próprios métodos e campos
- A classe de uma classe é a sua *metaclasse*; se uma classe é subclasse de outra, então a sua metaclasse é subclasse da metaclasse da outra
- Metaclasses não precisam ser objetos, mas se forem todos podem ser instâncias de uma única classe
- Cada classe do sistema é uma instância única (um *singleton*) de sua *metaclasse*

Listas em Ruby

- O programa ao lado é uma versão em Ruby do programa *classe* de dois slides atrás
- `imprime` e `map` são *class methods* da classe `Vazia`, e `apply` é um class method da classe Quadrado
- Notem que não instanciamos `Vazia` e `Quadrado`! Eles já são objetos que têm os métodos que precisamos (`imprime`, `map`, `apply`)
- `new` também é um *class method* que executa uma primitiva parecida com a `new` de *classe* e depois o método `initialize` do objeto recém-criado

```
class Vazia
  def Vazia.imprime() end
  def Vazia.map(f) self  end
end
class Cons
  def initialize(h, t)
    @h = h; @t = t
  end
  def imprime()
    puts(@h); @t.imprime()
  end
  def map(f)
    Cons.new(f.apply(@h),
             @t.map(f))
  end
end
class Quadrado
  def Quadrado.apply(x)
    x * x
  end
end

l1 = Cons.new(1,
          Cons.new(2, Vazia))
l2 = l1.map(Quadrado)
l1.imprime()
l2.imprime()
```

tracer em Ruby

- Com metaclasses e *class methods* conseguimos implementar uma versão da função `tracer` do slide 6, coisa que não podemos fazer em *classe*
- Os métodos definidos com self são outro modo de definir *class methods*
- `Class.new` é uma maneira de definir uma classe anonimamente, usando uma superclasse
- Super chama o *class method* `apply` na superclasse, que por sua vez chama o *class method* `apply` na subclasse, por recursão aberta

```
class Fat
  def self.apply(n)
    if n<2 then
      1
    else
      n * self.apply(n-1)
    end
  end
end

def tracer(f)
 Class.new(f) do
   def self.apply(x)
     puts(x)
     super(x)
   end
 end
end

puts(tracer(Fat).apply(5))
```