Técnicas Avanzadas de Programación – UTN – FRBA
2 cuatrimestre 2014
Parcial – Metaprogramación – MultiMethods

Dominio

Se desde construir un framework para permitir definir métodos con múltiple dispatch.

El envío de mensajes definido en Ruby es single dispatch, esto quiere decir que sólo se mira el tipo del objeto receptor para determinar que método se va a ejecutar (esto es lo que permite el polimorfísmo).

La idea es extender este mecanismo para que también se vean los parámetros para determinar la ejecución del método. De esta forma habrá distintas implementaciones del mismo mensaje en el mismo objeto, y se seleccionará una de ellas en base a la forma de los parámetros.

Requerimientos

1. Dispatching por tipo de parámetros

Considere la siguiente clase:

```
class StringUtils
  multimethod :concat do
    define_for [String, String] do |s1, s2|
      s1 + s2
    end
    define_for [String, Integer] do |s, n|
      s * n
    end
    define_for [Array] do |a|
      a.join
    end
  end
end
```

y considere los siguientes specs:

```
utils = StringUtils.new
expect(utils.concat('hola', 'mundo')).to eq('holamundo')
expect(utils.concat('hola', 3)).to eq('holaholahola')
expect(utils.concat(['hola', ' ', 'mundo'])).to eq('hola mundo')
```

Se desea poder definir métodos cuyo dispatch esté dado por los tipos de los parámetros que recibe al momento de ejecutarse. Tomando el ejemplo anterior, se desprende que para el mismo mensaje `concat` ejecutará distintos 'métodos' (o bloques).

Esta selección de métodos debe hacerse para cada envío de mensaje en base a los valores de los parámetros.

La selección del método a ejecutar usará como precedencia el orden en el que estuviesen definidos los métodos y quedará a cargo del usuario del framework escribirlos en el orden correcto.

Debe considerarse que todo objeto debe poder definir un multimétodo.

2. Dispatching por valores y procs

Se desea extender el mecanismo anterior para que puedan definirse los métodos no sólamente en base a los tipos de los parámetros, sino también para valores concretos y bloques arbitrarios.

De esta forma se puede elegir un método si tiene un valor en particular o si cumple con un bloque provistor por el usuario del framework.

Ejemplo:

```
class StringUtils
  multimethod :concat do
    define_for [nil] do |o|
      nil
    end
    define_for [String, -1] do |s, n|
      s.reverse
    end
    define_for [String, proc {|o| o.odd? or o == 42}] do |s, n|
      true
    end
    define_for [String, Integer] do |s, n|
      s * n
    end
  end
end
```

Specs:

```
utils = StringUtils.new
expect(utils.concat(nil)).to eq(nil)
expect(utils.concat('hola', -1)).to eq('aloh')
expect(utils.concat('hola', 45)).to eq(true)
expect(utils.concat('hola', 2)).to eq('holahola')
```

Los métodos se evaluarán en el orden en que fueron declarados.

NOTA: En Ruby, usar `proc`, `lambda` o `Proc.new` termina siempre creando un objeto de tipo `Proc`.

3. Dispatching por la forma del parámetro

En esta sección se pide definir los métodos en base a la forma del objeto. De esta manera se puede definir el parámetro por los mensajes que entiende. Para ello el framework deberá proveer el mensaje `duck` el cual recibirá la lista de selectores que el objeto deberá entender.

Ejemplo:

```
class Persona
  attr_accessor :nombre, :apellido
  def initialize
    @nombre = 'Johann Sebastian'
    @apellido = 'Mastropiero'
  end
end

class StringUtils
  multimethod :concat do
    define_for [String, duck(:nombre, :apellido)] do |s, p|
      "#{s} #{p.nombre} #{p.apellido}!"
    end
    define_for [String, String] do |s1, s2|
      s1 + s2
    end
    define_for [String, Integer] do |s, n|
      s * n
    end
  end
end
```

Specs:

```
utils = StringUtils.new
expect(utils.concat('hola', Persona.new)).to eq('Hola Johann Sebastian Mastropiero')
expect(utils.concat('hola', 'mundo')).to eq('hola mundo')
expect(utils.concat('hola', 2)).to eq('holahola')
```

4. Permitir definir multimethods a nivel de clase

Hasta este punto solo se está pudiendo definir los multimétodos para instancias de una clase. Se quiere poder extender esta funcionalidad para que también puedan definirse multimétodos a nivel de clase, respetando la siguiente sintaxis:

```
class StringUtils
  self_multimethod :concat do
    define_for [String, String] do |s1, s2|
      s1 + s2
    end
    define_for [String, Integer] do |s, n|
      s * n
    end
  end
end
```

Specs:

```
expect(StringUtils.concat('hola', 'mundo')).to eq('hola mundo')
expect(StringUtils.concat('hola', 2)).to eq('holahola')
```

5. BONUS: Soporte para herencia

Por último se pide soportar la herencia, dandole precedencia a las definiciones de las clases hijas por sobre las definiciones de la clase padre. No es necesario soportar el uso de super dentro de los métodos definidos como multimethods.

Ejemplo:

```
class StringUtils
  multimethod :concat do
    define_for [String, String] do |s1, s2|
      s1 + s2
    end
  end
end

class SuperStringUtils < StringUtils
  multimethod :concat do
    define_for [String, String] do |s1, s2|
      s2 + s1
    end

    define_for [String, Array] do |s, a|
      a.join s
    end
  end
end
```

Specs:

```
utils = StringUtils.new
super_utils = SuperStringUtils.new
expect(super_utils.concat('hola', 'mundo')).to eq('mundohola')
expect(super_utils.concat('-', ['hola', 'mundo'])).to eq('hola-mundo')
expect(utils.concat('hola', 'mundo')).to eq('holamundo')
```