Técnicas Avanzadas de Programación

Ejercicio Metaprogramación MultiMethods[1]

Introducción

En la mayoría de los lenguajes de programación orientados a objetos, el mecanismo de dispatch es simple: al momento de compilación se determina la firma del método, y al momento de ejecución se determina la implementación a ejecutar en función del receptor del mensaje. Esto permite lograr el polimorfismo: el método a ejecutar depende del tipo del receptor en ejecución.

Algunos lenguajes como Xtend soportan un mecanismo distinto, conocido como multiple dispatch o polimorfismo paramétrico, y básicamente permite determinar en tiempo de ejecución el método a ejecutar no sólo en función del receptor sino de los parámetros. Un conjunto de métodos definido con el mismo selector, pero con implementaciones para firmas que difieren es llamado multimethod.

El objetivo de este trabajo práctico es implementar en Ruby un framework que permita definir multimethods a nivel de módulo, llevando el multiple dispatch a Ruby.

1. Partial Blocks

Como primera aproximación, se desea contar con la posibilidad de definir un bloque de código que aplica o no a determinados argumentos. Para ello se pide implementar la clase PartialBlock, que pueda utilizarse de la siguiente manera:

```
helloBlock = PartialBlock.new([String]) do |who|
  "Hello #{who}"
end
5
helloBlock.matches?("a") #true
helloBlock.matches?(1) #false
helloBlock.matches?("a", "b") #false
```

Un bloque parcial se construye a partir de una lista de tipos y un bloque. El constructor debe validar que la lista de tipos tenga tantos elementos como argumentos tenga el bloque proporcionado.

Un bloque parcial bien construido debe ser capaz de validar si está definido para un conjunto de valores dado, es decir, si podría ser evaluado con esos valores cómo parámetro. Esto se dá si los tipos de los argumentos son los que se especificaron en la definición y si además son la misma cantidad que los argumentos.

Un bloque parcial debe poder ser ejecutado enviandole el mensaje *call*. Si los argumentos recibidos son válidos, retorna el resultado de ejecutar, sino, lanza un ArgumentError.

[1] Este enunciado es una versión acotada del Trabajo Práctico de Metaprogramación - 2015 1C MultiMethods, incluyendo también lo requerido como TP individual.

```
helloBlock.call("world!") #devuelve "Hello world!"
helloBlock.call(1) #Arroja excepción! no matchea el tipo
```

Cabe aclarar que un bloque puede ser ejecutado con instancias de subtipos de los que define y debe funcionar:

```
pairBlock = PartialBlock.new([Object, Object]) do |left, right|
  [left, right]
end

pairBlock.call("hello", 1) #Funciona (String e Integer extienden Object).
```

Tener en cuenta que los módulos también cuentan como tipos y deben ser soportados para usar como parte de la firma.

2. Multi Methods

El objetivo ahora es implementar multimétodos. Un módulo o clase debe poder definir un multimethod definiendo cada una de sus firmas cómo una *definición parcial*.

```
class A
  partial_def :concat, [String, String] do |s1,s2|
    s1 + s2
  end

  partial_def :concat, [String, Integer] do |s1,n|
    s1 * n
  end

  partial_def :concat, [Array] do |a|
    a.join
  end
end

A.new.concat('hello', ' world') # devuelve 'hello world'
A.new.concat('hello', 3) # devuelve 'hellohellohello'
A.new.concat(['hello', ' world', '!']) # devuelve 'hello world!'
A.new.concat('hello', 'world', '!') # Lanza una excepción!

A.multimethods() #[:concat]
A.multimethod(:concat) #Representación del multimethod
```

En este ejemplo, la clase A define un único multimethod *concat* con tres piezas de código asociadas a tres firmas distintas.

Cuando un objeto recibe un mensaje implementado con multimethods, el framework debe decidir cual implementación parcial ejecutar en función de los parámetros, de la siguiente forma:

1. Sólo debe considerar aquellas definiciones parciales que matcheen con los parámetros pasados (deben tener mismo tipo y aridad).

2. De estas definiciones, elige cual ejecutar realizando un cálculo de "distancia de parámetros" y quedándose con la definición que presente la distancia menor.
   La distancia se calcula en función del tipo de los parámetros y los tipos para los cuales está definido el bloque parcial, de la siguiente forma:
   Distancia Total de Parámetros = suma de la distancia de cada parámetro, multiplicada por el indice del parámetro (su posición en el array, empezando en 1)
   Distancia de Parámetro x = x.class.ancestors.index(tipoDefinidoEnBloqueParcial)

De esta manera, se busca ejecutar la definición más específica de todas las que coinciden.
Por ejemplo:

```
class A
  partial_def :concat, [String, Integer] do |s1,n|
    s1 * n
  end

  partial_def :concat, [Object, Object] do |o1, o2|
    "Objetos concatenados"
  end
end

A.new.concat("Hello", 2) # "HelloHello", ya que ("Hello", 2) está a menor distancia de [String,Integer] que de [Object,Object].

A.new.concat(Object.new, 3) # "Objetos concatenados" [Object,Object] es la única definición que aplica.
```

A su vez, se pide extender la interfaz de reflection con métodos que indiquen si un objeto responde a un determinado mensaje con cierta firma:

```
A.new.respond_to?(:concat) # true, define el método como multimethod
A.new.respond_to?(:to_s) # true, define el método normalmente
A.new.respond_to?(:concat, false, [String,String]) # true, los tipos coinciden
A.new.respond_to?(:concat, false, [Integer,A]) # true, matchea con [Object, Object]
A.new.respond_to?(:to_s, false, [String]) # false, no es un multimethod
A.new.respond_to?(:concat, false, [String,String,String]) # false, los tipos no coinciden
```

NOTA: *respond_to?* es un mensaje que ya viene implementado en Ruby y es **muy importante** que su funcionamiento actual se mantenga.
El mismo define dos parámetros, el primero indica el nombre del mensaje, el segundo un boolean que indica si debe inspeccionar también los métodos privados (con default en “false”).
http://ruby-doc.org/core-2.2.2/Object.html#method-i-respond_to-3F

Cabe aclarar que el código de los multimethods debe poder hacer referencia a self, que debe apuntar al objeto receptor del mensaje:

```
class Soldado
# ... implementación de soldado
end

class Tanque
  # ... implementación de tanque

  partial_def :ataca_a, [Tanque] do |objetivo|
    self.ataca_con_canion(objetivo)
  end

  partial_def :ataca_a, [Soldado] do |objetivo|
    self.ataca_con_ametralladora(objetivo)
  end
end
```

Así como con los métodos comunes, también, debe ser posible definir un multimethod para una clase ya existente, pero teniendo en cuenta que cada definición parcial se agrega a las anteriores (a menos que esté definida para una firma para la cual ya existe una definición parcial, en cuyo caso la nueva pisa la anterior):

```
class Tanque
  # ... implementación de tanque

  partial_def :ataca_a, [Tanque] do |objetivo|
    self.ataca_con_canion(objetivo)
  end

  partial_def :ataca_a, [Soldado] do |objetivo|
      self.ataca_con_ametralladora(objetivo)
  end
end

class Avion
  #... implementación de avión
end

#abro la clase tanque
class Tanque
  #Agrego una implementación para atacar aviones que NO pisa las anteriores
  partial_def :ataca_a, [Avion] do |avion|
    self.atacar_con_satelite(avion)
  end

  #Cambio la definición previa de cómo atacar a un soldado
  partial_def :ataca_a, [Soldado] do |soldado|
    self.pisar(soldado)
  end
end
```

3. Nuevo requerimiento

Se solicita extender la implementación actual de multimethods para soportar definiciones parciales que incluyan soporte para “duck typing” o tipado estructural. Esto es, en lugar de solamente permitir que en las firmas se incluyan clases o módulos, se desea permitir poder especificar **una lista de mensajes que el objeto debe entender** para encajar en la definición:

Para que un parámetro sea aceptado por una restricción de este tipo, el parámetro debe entender **TODOS** los mensajes pedidos, de lo contrario, la definición no aplica.

```
class A
  partial_def :formatear, [String, [:nombre, :direccion]] do |titulo, coso|
    titulo + " | " + coso.nombre + ": " + coso.direccion
  end
  partial_def :formatear, [String, [:peso]] do |titulo, pesable|
    titulo + " " pesable.peso
  end
end

class Lugar
    attr_accessor :nombre, :direccion, :fotos
    def initialize (...)
end

class Perro
    attr_accessor :nombre, :edad, :peso
    def initialize (...)
end

A.new.formatear("VISITE", Lugar.new("Obelisco", "Corrientes y 9 de Julio"))
#devuelve "VISITE | Obelisco: Corrientes y 9 de Julio"

A.new.formatear("Pesado", Perro.new(32))
#devuelve "Pesado 32"

A.new.formatear("Pesado", 5)
#error! 5 no matchea con ningún tipo posible"
```

Esta nueva forma de especificar el tipo de los parámetros debe ser compatible con la ya existente (se debe poder usar una o la otra indistintamente).

La distancia de una restricción de duck typing es siempre de 0.5 (es decir, solo la propia clase es más específica que una definición estructural).