Universidad “Simón Bolívar”

Departamento de Computación y Tecnología de la Información

CI3661 – Taller de Lenguajes de Programación I

Enero-Marzo 2015

Programación Orientada a Objetos

Unificación

En este ejercicio de programación, Ud. implantará el Algoritmo de Unificación de Prolog aplicando los conceptos de despacho doble y posiblemente *duck-typing* discutidos en clase. En su solución, no puede usar introspección, reflexión, variables globales, ni variables de clase: debe ser una solución **pura** orientada a objetos con despacho dinámico simple o doble, según convenga.

Con este propósito, modelaremos un conjunto simplificado de los tipos de datos provistos por Prolog usando Ruby.

Términos

En el sentido más estricto, Prolog manipula términos, que pueden ser atómicos, variables o functores. Entonces, es necesario que Ud. provea la clase abstracta `Term` que tenga las subclases `Atomic`, `Variable` y `Functor`, para representar los términos manipulables en el algoritmo.

Clase `Atomic` (0.5 puntos)

La clase `Atomic` representará cualquier átomo Prolog, sin importar si se trata de un número o un átomo simbólico. Por lo tanto, es necesario proveer un constructor que inicializará el símbolo, el cual no puede ser cambiado posteriormente

```
irb> atom42 = Atomic.new(42)
```

Además, es necesario proveer los métodos

- `value` – para leer el atributo correpondiente al símbolo.

  ```
  irb> atom42.value
  => 42
  ```

- `to_s` – para mostrar al invocante como un `String`

```
irb> atom42.to_s
=> "Atom 42"
```

Clase `Variable` (0.5 puntos)

La clase `Variable` representará una variable Prolog, sin importar si está escrita en mayúsculas o minúsculas. Por lo tanto, es necesario proveer un constructor que inicializará el nombre de la variable, el cual no puede ser cambiado posteriormente

```
irb> varx = Variable.new('x')
```

Además, es necesario proveer los métodos

- `name` – para leer el atributo correpondiente al símbolo.

```
irb> varx.name
=> "x"
```

- `to_s` – para mostrar al invocante como un `String`

```
irb> varx.to_s
=> "Var x"
```

Clase `Functor` (0.5 puntos)

La clase `Functor` representará un functor Prolog de aridad arbitraria. Por lo tanto, es necesario proveer un constructor que inicializará el nombre del functor, establecerá los argumentos iniciales usando un `Array` Ruby, atributos que no pueden ser cambiados posteriormente

```
irb> functorBar = Functor.new('bar',[ atom42, varx, varx ])
```

Puede suponer que el `Array` sólo contendrá elementos que son **subclase** de `Term`.

Además, es necesario proveer los métodos

- `name` – para leer el atributo correpondiente al nombre

```
irb> functorBar.name
=> "bar"
```

- `args` – para tener acceso al `Array` de argumentos

  ```
  irb> functorBar.args
  => [Atom 42, Var x, Var x]
  ```

- `to_s` – para mostrar al invocante como un `String`, usando una presentación similar a la de Prolog

  ```
  irb> functorBar.to_s
  => "Functor bar(Atom 42,FVar x,FVar x)"
  ```

Algoritmo de Unificación (3 puntos)

Todas las clases antes mencionadas deben proveer el método `unify` que implante el Algoritmo de Unificación con *Occurs Check*. El método se invocará

```
term0.unify(term1)
```

donde `term0` y `term1` podrían ser cualquie combinación de objetos de las clases `Atomic`, `Variable` o `Functor`. Si la unificación es posible, el método debe retornar un `Hash` de Ruby indicando las asignaciones de variable necesarias por ejemplo

```
irb> atom42.unify(varx)
=> {Var x=>Atom 42}
```

Note que si no hay variables se retornaría un `Hash` vacío.

```
irb> atom42.unify(atom42)
=> {}
```

Si la unificación no es posible, el método debe retornar un objeto `nil`.

```
irb> atom42.unify(atom69)
=> nil
```

Si el invocante o el argumento son de la clase `Term`, debe emitirse una excepción – después de todo, la clase `Term` es abstracta.

En su implantación del método `unify` **no** puede utilizar introspección, i.e. **no** puede usar ninguno de los métodos de la clase `Class` para determinar el tipo particular del invocante o del argumento. **Sólo** puede usar despacho doble, así que necesitará definir métodos adicionales para lograrlo.

Además, note que por la naturaleza del Algoritmo de Unificación, la llamada inicial es de la forma

```
term0.unify(term1)
```

pero en las llamadas recursivas seguramente tendrá que invocar

```
foo.unify(bar,ambiente)
```

donde `foo` y `bar` son términos intermedios que aparecen en el cuerpo de su implantación, y `ambiente` es el diccionario de variables asignadas hasta ese punto. Sólo puede haber un método `unify` en sus clases, de manera que debe aprovechar las capacidades de Ruby para simplificar la definición de métodos con número variables de argumentos.

Durante su implantación del Algoritmo de Unificación, notará que necesita hacer la diferencia entre una variable libre y una variable ligadad. Hay varias maneras de resolver ese problema, pero en virtud de las limitaciones impuestas para el ejercicio, con el propósito de que Ud. utilice **exclusivamente** técnicas puras de Orientación a Objetos, las dos maneras directas de manejar el problema son:

- Tener la clase `BoundedVariable` subclase de `Variable`, que establezca la diferencia entre una variable ligada y una variable libre.
- Usar estructuras adicionales **internas** (variables locales o argumentos) en los métodos para unificación.

Si Ud. encuentra una solución alternativa a este problema, por favor consúltela antes de entregar, para asegurar que se trata de una solución pura desde el punto de vista Orientado a Objetos.

Búsqueda Generalizada

Arboles y grafos explícitos (1 punto)

Considere la siguiente definición para una clase que representa árboles binarios

```
class BinTree
  attr_accessor :value,  # Valor almacenado en el nodo
                :left,   # BinTree izquierdo
                :right   # BinTree derecho
  def initialize(v,l,r)
    # Su código aquí
  end
  def each(&block)
    # Su código aquí
  end
end
```

en la cual el propósito del método `each` es recibir un bloque que será utilizado para iterar sobre los hijos del nodo, cuando estén definidos.

Ahora, considere la siguiente definición para una clase que representa grafos arbitrarios a partir de un nodo específico, indicando sus sucesores

```
class GraphNode
  attr_accessor :value,   # Valor alamacenado en el nodo
                :children # Arreglo de sucesores GraphNode
  def initialize(v,c)
    # Su código aquí
  end
  def each(&block)
    # Su código aquí
  end
end
```

en la cual el propósito de `each` es recibir un bloque que será utilizado para iterar sobre los hijos del nodo, cuando estén definidos.

Recorrido BFS como comportamiento (3 puntos)

Se desea que Ud. implante una infraestructura de búsqueda BFS [1] que pueda ser utilizada por ambas clases, empleando la técnica de *mixins* estudiada en clase. Se espera que su *mixin* ofrezca la siguiente interfaz de programación

- `find(start,predicate)` que comienza la búsqueda BFS a partir del objeto `start` hasta encontrar el primer objeto que cumpla con el predicado `predicate`, y lo retorna. Si se agota la búsqueda sin encontrar objetos que cumplan el predicado, se retorna el objeto `nil`.
- `path(start,predicate)` que comienza la búsqueda BFS a partir del objeto `start` hasta encontrar el primer objeto que cumpla con el predicado `predicate`, y retorna el camino desde `start` hasta el nodo encontrado, en forma de `Array` de objetos. Si se agota la búsqueda sin encontrar objetos que cumplan el predicado, se retorna el objeto `nil`.
- `walk(start,action)` que comienza un recorrido BFS a partir del objeto `start` hasta agotar todo el espacio de búsqueda, ejecutando el cuerpo de código `action` sobre cada nodo visitado y retornando un `Array` con los nodos visitados. Si el cuerpo de código `action` se omite, sólo debe retornar el `Array` con los nodos visitados.

Para la implantación de su *mixin* sólo puede suponer que las clases usuarias disponen del método de acceso `value` para obtener el valor almacenado en un

nodo, y del método `each` para recorrer todos los hijos de un nodo particular. Los métodos de su *mixin* no pueden crear variables de instancia ni de clase adicionales. Seguramente necesitará métodos adicionales dentro de su mixin para asistirlo en la implantación de su recorrido BFS.

Arboles implícitos (1.5 puntos)

Una de las clases sobre Prolog versó sobre la resolución de problemas de búsqueda utilizando un árbol implícito de expansión, incluyendo recortes inteligentes para reducir la complejidad en tiempo y espacio. En particular estudiamos el problema del "Lobo, Cabra y Repollo".

Suponga la siguiente definición de clase que permite modelar un estado del problema

```
def LCR
  attr_reader :value
  def initialize(?)  # Indique los argumentos
    # Su código aquí
  end
  def each(p)
    # Su código aquí
  end
  def solve
    # Su código aquí
  end
end
```

en la cual:

- El atributo `value`, que sólo se puede leer, corresponde a un estado particular del problema representado como un `Hash`. El `Hash` tiene tres claves `where`, `left` y `right`, para indicar la posición del bote con un valor de la clase `Symbol`, así como los contenidos de las orillas izquierda y derecha usando `Array`s de `Symbol`.
- El método `each` recibe un bloque que será utilizado para iterar sobre los hijos del estado actual. Los hijos deben ser generados dinámicamente, y el cuerpo de código sólo debe iterar sobre aquellos que tengan sentido en el contexto del problema de búsqueda.
- El método solve resuelve el problema de búsqueda

```
initial = LCR.new(...)
initial.solve()
```

... salen en pantalla los movimientos necesarios...

La clase `LCR` debe aprovechar el mixin de BFS para resolver el problema.

Entrega de la Implementación

Ud. debe entregar un archivo .tar.gz o .tar.bz2 (no puede ser .rar, ni .zip) que al expandirse genere un directorio con el nombre de su grupo (e.g. G42) dentro del cual encontrar solamente:

- Los archivos `unify.rb` y `bfs.rb` conteniendo el código fuente Ruby para implantar las clases y mixins correspondientes a cada sección.
- Se evaluará el correcto estilo de programación empleando una indentación adecuada y consistente en cualquier editor ("profe, en el mío se ve bien" es inaceptable).
- El archivo debe estar completa y correctamente documentado [2]. Es inaceptable que la documentación tenga errores ortográficos.
- Su programa será corregido usando Ruby 1.9 empleando exclusivamente las librerías estándar incluidas con la distribución. No está permitido utilizar librerías externas ni gemas.
- Puede escribir métodos adicionales si los necesita, pero estos no pueden ser públicos.
- Fecha de Entrega. Viernes 2015-03-13 hasta las 23:59 VET
- Valor de Evaluación. Diez (10) puntos.

Referencias

Referencias

[1] BFS en Wikipedia

[2] Ruby Documentation System