Laboratorio de Programación Multiparadigma (LPM)

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

---

Práctica 3: Programación lógico-funcional en Maude

Santiago Escobar

Julia Sapiña

Índice

1. Introducción

En esta parte vamos a describir el lenguaje de programación de alto rendimiento Maude y, a continuación, mostraremos cómo realizar programación lógico-funcional en Maude.

2. El lenguaje de programación Maude

El lenguaje de programación Maude utiliza reglas de reescritura, como los lenguajes denominados funcionales tales como Haskell, ML, Scheme, o Lisp. En concreto, el lenguaje Maude está basado en la lógica de reescritura que permite definir multitud de modelos computacionales complejos tales como programación concurrente o programación orientada a objetos. Por ejemplo, Maude permite especificar objetos directamente en el lenguaje, siguiendo una aproximación declarativa a la programación orientada a objetos que no está disponible ni en lenguajes imperativos como C++ o Java ni en lenguajes declarativos como Haskell.

El desarrollo del lenguaje Maude parte de una iniciativa internacional cuyo objetivo consiste en diseñar una plataforma común para la investigación, docencia y aplicación de los lenguajes declarativos. Se puede encontrar más información en:

`http://maude.cs.uiuc.edu`

A continuación resumimos las principales características del lenguaje Maude. Sin embargo, hay un extenso manual y un "*primer*" (libro de introducción muy sencillo y basado en ejemplos) en la dirección web indicada antes. Existe también un libro sobre Maude, con ejemplares adquiridos por la Biblioteca General y la ETSInf, y accesible online en:

`http://www.springerlink.com/content/p6h32301712p`

Es interesante destacar que el manual de Maude accesible desde la página oficial de Maude indicada arriba contiene actualmente una sección dedicada a la programación lógico-funcional en Maude.

2.1. Un programa en Maude

Un programa Maude está compuesto por diferentes módulos. Cada módulo se define entre las palabras reservadas `mod` y `endm`, si es un *módulo de sistema*, o entre `fmod` y `endfm`, si es un *módulo funcional*. En esta práctica nos vamos a restringir sólo a módulos de sistema.

Cada módulo incluye declaraciones de tipos y símbolos, junto con las reglas, encabezadas por `rl`, que describen la lógica de algunos de los símbolos, llamadas *funciones*. Básicamente, los símbolos y reglas definidos en un módulo de sistema tienen un comportamiento indeterminista y ejecuciones posiblemente infinitas en el tiempo (es decir, que no terminen nunca). Por ejemplo, el siguiente módulo de sistema simula una máquina de café y galletas:

```
mod VENDING-MACHINE is
  sorts Coin Coffee Cookie Item State .
  subsorts Coffee Cookie < Item .
  subsorts Coin Item < State .

  op null : -> State .
  op __ : State State -> State [assoc comm id: null] .
  op $ : -> Coin .
  op q : -> Coin .
```

```
  op a : -> Cookie .
  op c : -> Coffee .

  var St : State .

  rl St => St q .   --- Modela que se ha anyadido un cuarto de dolar
  rl St => St $ .   --- Modela que se ha anyadido un dolar
  rl $ => c .       --- Modela que se ha tragado el dolar y ha devuelto un cafe
  rl $ => a q .     --- Devuelve una galleta y un cuarto de dolar
  rl q q q q => $ . --- Cambia cuatro cuartos de dolar por un dolar
endm
```

Este sistema es indeterminista (p.ej. para un dólar "$" hay dos posibles acciones) y no terminante (siempre se puede añadir más dinero a la máquina).

Las reglas tienen un significado indeterminista, por lo tanto el comando `rewrite` escoge una regla al azar cuando hay más de una posibilidad y nos devuelve una de las posibles ejecuciones del programa. En el caso de tener un conjunto de reglas que no terminan, no podremos lanzar el comando `rewrite` y deberemos usar el comando `search`, que nos indica si en el espacio de posibles ejecuciones del programa existe una ejecución que satisfaga ciertas condiciones. Por ejemplo, dado el módulo `VENDING-MACHINE` mostrado anteriormente y asumiendo que está almacenado en el fichero `vending.maude` escribiremos

```
$ maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
           Maude 2.4 built: Nov 12 2009 18:47:47
             Copyright 1997-2009 SRI International
                  Tue Nov 24 12:47:11 2009
Maude> load vending.maude
Maude> search [1] $ =>* c c .
search [1] in VENDING-MACHINE : $ =>* c c .
Solution 1 (state 22)
states: 23  rewrites: 79 in 0ms cpu (10ms real) (~ rewrites/second)
empty substitution

Maude> show path 22 .
state 0, Coin: $
===[ rl St => $ St . ]===>
state 2, State: $ $
===[ rl $ => c . ]===>
state 10, State: $ c
===[ rl $ => c . ]===>
state 22, State: c c
Maude> quit
Bye.
```

donde buscamos si es posible ir de un estado con un dólar a un estado con dos cafés. En este caso tenemos que restringir la búsqueda a la primera solución (parte `[1]` del comando) ya que el espacio

de búsqueda es infinito. Nótese que el comando "`show path N .`" muestra la solución encontrada en el proceso de búsqueda, donde N es el número con el que Maude identifica el estado del espacio de búsqueda asociado a cada solución, es decir, "`Solution i (state N)`".

3. Programación lógico-funcional en Maude

La programación lógico-funcional en Maude se basa en el mecanismo operacional denominado *narrowing* (o estrechamiento). Las *expresiones de respuesta* de narrowing se denotan por pares $< e, \sigma >$, donde σ es la sustitución computada y `e` es el valor de retorno final. Por ejemplo, dada la función

```
rl f(a) => b .
rl f(c) => d .
```

el resultado de evaluar la expresión `f(c)` es `d`, mientras que la evaluación de `f(x)`, donde `x` es una variable, devuelve dos expresiones de respuesta:

```
< b , x <-- a >
< d , x <-- c >
```

Cuando se produce una llamada a función cuyos argumentos no estan lo suficientemente instanciados, se instancian las variables de forma que la llamada a función se pueda resolver usando las distintas reglas que definen la función (es decir, empleando *narrowing*), lo que dará lugar a más de un resultado final (como en el caso de `f(x)`).

A partir de la version 3.0 de Maude, existe un comando `vu-narrow` para narrowing. Podemos reproducir el ejemplo anterior y su salida.

```
mod TEST is
   sort A .
   ops a b c d : -> A .
   op f : A -> [A] .
   vars X Y Z : A .
   rl f(a) => b [narrowing] .
   rl f(c) => d [narrowing] .
endm

Maude> vu-narrow f(X) =>* Y .

Solution 1
state: b
accumulated substitution:
X --> a
variant unifier:
Y --> b

Solution 2
state: d
accumulated substitution:
X --> c
```

```
variant unifier:
Y --> d

No more solutions.
```

El nuevo comando tiene algunas restricciones de uso y notas adicionales:

- Las reglas usadas para narrowing deben llevar la etiqueta `narrowing`.
- No se admiten reglas condicionales.
- Las ecuaciones existentes en los módulos utilizados por narrowing son aplicadas sólo por reescritura, es decir, no se instancian las variables y sirven sólo para simplificar términos. Existen ecuaciones para hacer unificación ecuacional pero no se presentan en esta práctica.
- Asímismo, todas las funciones primitivas de Maude usadas en los módulos utilizados por narrowing son aplicadas sólo por reescritura, es decir, no se instancian las variables de dichas funciones primitivas.
- Los módulos admitidos solo pueden contener símbolos con las siguientes combinaciones de axiomas:
  - ningún axioma;
  - asociatividad (`assoc`);
  - conmutatividad (`comm`);
  - conmutatividad e identidad (`comm id`),
  - conmutatividad y asociatividad (`assoc comm`),
  - identidad y por derecha o izquierda (`id`, `left id`, `right id`),
  - conmutatividad, asociatividad, e identidad (`assoc comm id`).

  La única combinación que queda excluída es asociatividad e identidad (`assoc id`).
- `vu-narrow [<limit1>,<limit2>] <expresion1> =>* <expression2> .` Se busca algún camino de ejecución que lleve de una instancia del término `<expression1>` a una instancia del término `<expression2>`. Ambos términos pueden contener variables. La parte "`=>*`" se puede sustituir por "`=>!`" indicando la forma normal fuerte (es decir, cuando ya no queden más pasos de narrowing por hacer). De forma opcional, se busca varios caminos de ejecución, hasta un máximo de `<limit1>` soluciones y con un máximo de `<limit2>` pasos de narrowing. Se puede escribir `[,<limit2>]` si no deseamos un límite de soluciones.

4. Objetivo de la práctica

El objetivo de esta práctica consiste en codificar los ejercicios adjuntos en Maude y responder a las preguntas.

4.1. Monty Python Example

Dada la siguiente especificación del ejemplo de Monty Python en programación lógica visto en clase

```
bruja(X) :- arde(X), mujer(X) .
arde(X) :- madera(X) .
madera(X) :- flota(X) .
flota(X) :- mismopeso(pato, X) .
mujer(lola) .
mismopeso(lola,pato) .
mismopeso(pato,jamon) .
```

Una forma de codificarlo en Maude, usando narrowing y unificación, es la siguiente:

```
mod BRUJA is
  sort A .
  ops pato lola jamon : -> A .

  var X : A .

  op bruja : A -> [Bool] .
  rl bruja(X) => arde(X) and mujer(X) [narrowing] .

  op mujer : A -> [Bool] .
  rl mujer(lola) => true [narrowing] .

  op arde : A -> [Bool] .
  rl arde(X) => madera(X) [narrowing] .

  op madera : A -> [Bool] .
  rl madera(X) => flota(X) [narrowing] .

  op flota : A -> [Bool] .
  rl flota(X) => mismopeso(pato,X) [narrowing] .

  op mismopeso : A A -> [Bool] .
  rl mismopeso(lola,pato) => true [narrowing] .
  rl mismopeso(pato,jamon) => true [narrowing] .

endm
```

Y las preguntas que hay que responder son

1. ¿Qué devuelve la ejecución de la expresión `vu-narrow bruja(X) =>* true .`?
2. ¿Qué devuelve la ejecución de la expresión `vu-narrow bruja(lola) =>* true .`?
3. ¿Qué devuelve la ejecución de la expresión `vu-narrow bruja(lola) ~>* true .` si cambiamos la segunda regla de `mismopeso` para que compare `pato` y `lola`?

4.2. Listas

Dada la siguiente definición de la función `length` para calcular la longitud de una lista,

```
mod LISTAS is
  protecting NAT .

  sort NatList .
  op nil : -> NatList .
  op _:_ : Nat NatList -> NatList .

  var N : Nat .
  var NL : NatList .

  op length : NatList -> [Nat] .
  rl length(nil) => 0 [narrowing] .
  rl length(N : NL) => 1 + length(NL) [narrowing] .

endm
```

responde a las siguientes preguntas:

1. ¿Qué devuelve la ejecución de la expresión `rewrite length(nil) .`?
2. ¿Qué devuelve la ejecución de la expresión `vu-narrow length(1 : 2 : 3 : 4 : nil) =>! N .`?
3. ¿Qué devuelve la ejecución de la expresión `vu-narrow [1] length(NL) =>* 0 .`?
4. ¿Qué devuelve la ejecución de la expresión `vu-narrow [1] length(NL) =>* 10 .`?

4.3. Suma y comparación

Dada la siguiente definición de la función `sum` para calcular la suma de dos números enteros y la función `leq` para compara dos números enteros,

```
mod SUM-LEQ is
  protecting NAT .

  vars X Y : [Nat] .

  op sum : Nat Nat -> [Nat] .
  rl sum(0,Y) => Y [narrowing] .
  rl sum(s(X),Y) => s(sum(X,Y)) [narrowing] .

  op leq : Nat Nat -> [Bool] .
  rl leq(0,Y) => true [narrowing] .
  rl leq(s(X),s(Y)) => leq(X,Y) [narrowing] .

  op _=:=_ : [Nat] [Nat] -> [Bool] .
  rl X =:= X => true [narrowing] .
```

```
endm
```

responde a las siguientes preguntas:

1. ¿Qué devuelve la ejecución de la expresión `vu-narrow [1] leq(sum(X,Y),1) =>* true .`?
2. ¿Qué devuelve la ejecución de la expresión `vu-narrow [1] leq(sum(0,0),s(Y)) =>* true .`?
3. ¿Qué devuelve la ejecución de la expresión `vu-narrow [1] leq(sum(X,Y),s(X)) =>* true .`?
4. ¿Qué devuelve la ejecución de la expresión `vu-narrow [1] sum(X,Y) =:= 1 and leq(X,Y) =>* true .`?

4.4. Resta

¿Cómo especificarías en Maude una función que resta dos números naturales?

1. ¿Qué devuelve la expresión `vu-narrow [1] minus(s(0),s(s(0))) =>* X:Nat .`?
2. ¿Qué devuelve la expresión `vu-narrow [1] minus(s(s(0)),s(0)) =>* X:Nat .`?
3. ¿Qué devuelve la expresión `vu-narrow [1] minus(X:Nat,s(s(0))) =>* Y:Nat .`?