Laboratorio de Programación Multiparadigma (LPM)

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

---

Práctica 2: Programación y búsqueda 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 de la Escuela, y accesible online en:

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

2.1. Un programa en Maude

Un programa Maude está compuesto de 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*. Cada módulo incluye declaraciones de tipos y símbolos, junto con las reglas, encabezadas por `rl`, y las ecuaciones, encabezadas por `eq`, además de axiomas para listas o conjuntos, descritas con las palabras `assoc`, `comm`, e `id`. Las reglas describen transiciones entre distintas configuraciones (o estados) del modelo mientras que las ecuaciones y los axiomas describen cómo se ejecutan algunos de los símbolos, llamados *funciones*. En un módulo de sistema podremos incluir reglas y ecuaciones, pero en un módulo funcional sólo pueden aparecer ecuaciones.

Por ejemplo, podemos especificar el siguiente módulo funcional (sin reglas) que simula la función factorial:

```
fmod FACT is
  protecting INT .
  op _! : Int -> Int .
  var N : Int .
  eq 0 ! = 1 .                    --- factorial de N=0 es 1
  eq N ! = (N - 1)! * N [owise] . --- factorial de N>0 es N*factorial de N-1
endfm
```

Este sistema es determinista y termina para cada posible ejecución. Una ejecución asociada a este programa del factorial es la siguiente (donde subrayamos el subtérmino reemplazado):

$$\underline{4!} \longrightarrow \underline{3!} * 4 \longrightarrow \underline{2!} * 3 * 4 \longrightarrow \underline{1!} * 2 * 3 * 4 \longrightarrow \underline{1 * 2 * 3 * 4} \longrightarrow 24$$

Por otro lado, podemos especificar un módulo de sistema con reglas que representa una máquina de refrescos con distintas transiciones, incluyendo indeterminismo y cadenas infinitas de reescritura.

```
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).

2.2. Entorno de Trabajo: el sistema Maude

El sistema Maude no dispone en la actualidad de un compilador a código máquina, aunque existen varias versiones en pruebas. Sólo se dispone de un intérprete que se carga con el comando:

```
$ maude
```

mostrando el siguiente texto de presentación:

```
           \||||||||||||||||||/
         --- Welcome to Maude ---
           /||||||||||||||||||\
   Maude 2.7.1 built: Jun 27 2016 16:43:23
    Copyright 1997-2016 SRI International
         Mon Feb 13 10:17:23 2017
Maude>
```

El sistema está disponible para diferentes versiones de UNIX (incluyendo Linux y Mac OS X). Existe, además, una versión de Maude para Windows descargable siguiendo el enlace "Maude for Windows" de la página `http://safe-tools.dsic.upv.es/maude/`. Una vez arrancado el sistema, disponemos de los siguientes comandos:

`load < name > .` Lee y carga los distintos módulos almacenados en el archivo `<name>`. Los archivos de Maude se suelen crear con la extensión `.maude`.

`show modules .` Muestra los módulos cargados actualmente en el sistema.

`show module` < `module` > `.` Muestra el módulo `<module>` en pantalla.

`select` < `module` > `.` Selecciona un nuevo módulo para ser el módulo actual de ejecución de expresiones.

`reduce` < `expression` > `.` Evalúa la expresión `<expression>` con respecto al módulo actual usando sólo ecuaciones. También puede escribirse "`red` < `expression` > .".

`reduce in` < `module` > : < `expresion` > `.` Evalúa la expresión `<expression>` con respecto al módulo `<module>`.

`rewrite` < `expression` > `.` Evalúa la expresión `<expression>` con respecto al módulo actual usando ecuaciones y reglas. También puede escribirse "`rew` < `expression` > .".

`rewrite in` < `module` > : < `expresion` > `.` Evalúa la expresión `<expression>` con respecto al módulo `<module>`.

`search` < `expresion1` > `=>*` < `expression2` > `.` Busca algún camino de ejecución que lleve del término `<expression1>` al término `<expression2>`. El término `<expression2>` puede tener variables mientras que `<expression1>` no. La expresión "`=>*`" puede entenderse como buscar un camino de ejecución de `<expression1>` a `<expression2>` utilizando tantos pasos de ejecución como sean necesarios. También se puede sustituir por "`=>!`" indicando hasta que no sea posible dar más pasos de ejecución; o por "`=>1`" indicando un único paso de ejecución.

`search in` < `module` > : < `expresion1` > `=>*` < `expression2` > `.` Busca algún camino de ejecución con respecto al módulo `<module>`.

`search` [< `limit` >] < `expresion1` > `=>*` < `expression2` > `.` Busca varios caminos de ejecución, hasta un máximo de `<limit>`. Esto permite verificar propiedades de alcanzabilidad de forma positiva en sistemas con un número infinito de estados. **Cuidado** porque si no hay ninguna solución y el espacio de búsqueda es infinito el sistema no parará.

`cd` < `dir` > Permite cambiar de directorio (**cuidado** porque no lleva espacio ni punto al final).

`ls` Ejecuta el comando UNIX `ls` y muestra todos los ficheros en el directorio actual (**cuidado** porque no lleva espacio ni punto al final)

`quit` Salir del sistema (**cuidado** porque no lleva espacio ni punto al final).

La semántica operacional de Maude se basa en la lógica de reescritura y básicamente consiste en reescribir la expresión de entrada usando las reglas y ecuaciones del programa hasta que no haya más posibilidad. Para la ejecución de ecuaciones Maude utiliza una estrategia de ejecución *impaciente*, como ocurre en los lenguajes de programación imperativos como C o Pascal y en algunos lenguajes funcionales como ML, en vez de una estrategia de ejecución *perezosa*, como en el lenguaje funcional Haskell.

Por ejemplo, dada la función `_!` siguiente:

```
fmod FACT is
  protecting INT .
  op _! : Int -> Int .
  var N : Int .
```

```
  eq 0 ! = 1 .                    --- factorial de N=0 es 1
  eq N ! = (N - 1)! * N [owise] . --- factorial de N>0 es N*factorial de N-1
endfm
```

y asumiendo que está almacenada en el fichero `fact1.maude` escribiremos

```
$ maude
           \||||||||||||||||||/
         --- Welcome to Maude ---
           /||||||||||||||||||\
   Maude 2.7.1 built: Jun 27 2016 16:43:23
    Copyright 1997-2016 SRI International
         Mon Feb 13 10:17:23 2017
Maude> load fact1.maude
Maude> red 4 ! .
reduce in FACT : 4 ! .
rewrites: 13 in 0ms cpu (0ms real) (481481 rewrites/second)
result NzNat: 24
Maude> quit
```

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 2.7.1 built: Jun 27 2016 16:43:23
    Copyright 1997-2016 SRI International
   Mon Feb 13 10:17:23 2017
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 cada solución, es decir, "`Solution i (state N)`".

3. Objetivo de la práctica

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

3.1. Monty Python Example

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

```
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 sin utilizar narrowing, sólo usando reescritura, es la siguiente:

```
mod BRUJA-SAT is
  sort A .
  ops pato lola jamon : -> A .
  op _=:=_ : A A -> Bool .

  vars X Y : A .

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

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

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

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

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

  op mismopeso : A A -> [Bool] .
  rl mismopeso(X,Y) => X =:= lola and Y =:= pato .
  rl mismopeso(X,Y) => X =:= pato and Y =:= jamon .
```

```
endm
```

Y las preguntas que hay que responder son

1. ¿Qué devuelve la ejecución de la expresión `search bruja(X) =>* X:Bool .`?
2. ¿Qué devuelve la ejecución de la expresión `search bruja(lola) =>* X:Bool .`?
3. ¿Qué devuelve la ejecución de la expresión `search bruja(lola) =>* X:Bool .` si cambiamos la segunda regla de `mismopeso` para que compare `pato` y `lola`?

3.2. 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 .

  vars N N1 N2 : Nat .
  var NL : NatList .

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

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 `search length(1 : 2 : 3 : 4 : nil) =>! N .`?
3. ¿Qué devuelve la ejecución de la expresión `search length(1 : N1 : N2 : nil) =>! N .`?

3.3. 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] . var B : [Bool] .

  op sum : Nat Nat -> [Nat] .
```

```
  rl sum(0,Y) => Y .
  rl sum(s(X),Y) => s(sum(X,Y)) .

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

endm
```

responde a las siguientes preguntas:

1. ¿Qué devuelve la ejecución de la expresión `search leq(sum(0,0),s(Y)) =>* true .`?
2. ¿Qué devuelve la ejecución de la expresión `search leq(0,sum(X,Y)) =>* true .`?

3.4. Resta

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

1. ¿Qué devuelve la ejecución de la expresión `search minus(s(0),s(s(0))) =>* X:Nat .`?
2. ¿Qué devuelve la ejecución de la expresión `search minus(s(s(0)),s(0)) =>* X:Nat .`?
3. ¿Qué devuelve la ejecución de la expresión `search minus(0,s(X)) =>* Y:Nat .`?