Métodos Formales Industriales (MFI)

—prácticas—

Grado de Ingeniería en Informática

Práctica 1: Programación en Maude

Santiago Escobar

Despacho 237
Edificio DSIC 2 piso

1. Introducción

En esta práctica intentamos conseguir que el alumno refuerce en el laboratorio los conceptos introducidos sobre el lenguaje de especificación algebraico Maude durante las sesiones de teoría.

Los métodos formales en la Ingeniería Informática se definen sobre cuatro pilares básicos: (1) la existencia de un *modelo semántico* claro y conciso sobre el comportamiento de los sistemas software, (2) la existencia de una *representación* clara, detallada y sin ambigüedades que permita expresar sistemas software y asociarles una semántica concreta, (3) la existencia de un formalismo claro y detallado en el que se puedan definir *propiedades* de un sistema y en el que se pueda averiguar la validez o falsedad de dichas propiedades en función del modelo semántico y (4) la existencia de *técnicas eficientes y eficaces* para verificar dichas propiedades.

Los lenguajes de programación (o especificación) denominados declarativos son muy adecuados para ser utilizados durante el proceso de producción de software, especialmente en la definición de especificaciones prototípicas ejecutables, donde compiten con ventaja contra otro tipo de lenguajes de programación más complicados y con una semántica poco clara, como C#, Java, o Python.

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):

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

2.2. Errores comunes en los programas en Maude

- No finalizar las declaraciones con un espacio y un punto

  ```
  sort Natural
  op 0 : -> Natural.
  op s : Natural -> Natural
  ```

- no dejar espacios entre los argumentos y un operador infijo (o mixfix), p.ej. "0+0" es un error y hay que escribir "0 + 0"
- no incluir paréntesis para desambiguar expresiones, p.ej. "p s s 0 + 0" es un error y hay que escribir "p s s (0 + 0)" ó "(p s s 0) + 0" en función de la sintaxis apropiada.

2.3. Tipos de datos predefinidos

Maude dispone de varios tipos de datos predefinidos incluidos en el fichero `prelude.maude` de la instalación. En concreto, se dispone del tipo `Bool` definido en el módulo `BOOL`, el tipo `Nat` en el módulo `NAT`, el tipo `Int` en el módulo `INT`, el tipo `Float` en el módulo `FLOAT`, y los tipos `Char` y `String` en el módulo `STRING`. Para hacer uso de alguno de esos tipos, sus operadores, deberemos importar el módulo donde se encuentran con una de las palabras reservadas[1] `including`, `protecting` o `extending`. Por ejemplo, el módulo `FACT` para factorial mostrado anteriormente importa el módulo `INT` de los números enteros. El módulo `BOOL` es importado, por defecto, por todos los módulos cargados; esto se puede desactivar en el comando de llamada de Maude.

2.4. Declaración obligatoria de variables

Es obligatorio declarar el tipo de las variables antes de ser usadas en el programa, p.ej. la siguiente declaración de variables

```
vars N M : Nat .
var NL : NatList .
var NS : NatSet .
```

o añadirles el tipo directamente a las variables cuando vamos a usarlas, p.ej. "`X:Nat + Y:Nat`" .

[1] El significado de estas acciones para importar módulos queda fuera del objetivo principal de esta práctica, pero se puede consultar en el manual. La palabra reservada `protecting` es la más segura y simple.

2.5. Declaraciones de Tipos (Sorts), Símbolos Constructores y Variables

Una declaración de tipo tiene la forma:

```
sort T .
```

e introduce un nuevo tipo de datos `T`. Si queremos introducir varios tipos a la vez, escribiremos

```
sorts T1 ··· Tn .
```

Después se definen los constructores que formarán los datos asociados a ese tipo de la forma:

```
op C : T1 T2 ··· Tn -> T .
```

donde $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n$ son los tipos de los parámetros de ese símbolo. También podremos escribir

```
ops C1 ··· Cn : T1 T2 ··· Tn -> T .
```

y denota que todos los símbolos $\mathtt{C}_1, \ldots, \mathtt{C}_n$ tienen el mismo tipo. Por ejemplo, las declaraciones de tipo:

```
sort Bool .
ops true false : -> Bool .
sort NatList .
op nil : -> NatList .
op _:_ : Nat NatList -> NatList .
```

introducen el tipo `Bool` con dos constantes `true` y `false`, y el tipo `NatList` (listas cuyos elementos son naturales, es decir de tipo `Nat`). Hay que tener en cuenta que Maude no soporta tipos de datos polimórficos, en el sentido de Haskell, por lo tanto no se pueden definir listas polimórficas, aunque sí se puede definir módulos paramétricos con estructuras de datos dentro, disponiendo de listas paramétricas. Sin embargo, es interesante fijarse en la forma de definir el operador clásico binario infijo de construcción de una lista, "`_:_`", donde se indica que el primer argumento debe aparecer antes de los dos puntos mientras que el segundo detrás de los dos puntos. Una lista de enteros se podrá definir por lo tanto en notación infija como `0 : (1 : (2 : nil))` en vez de la notación prefija `:(0,:(1,:(2,nil)))` simplemente indicando que el símbolo a utilizar es "`_:_`". Esto es muy práctico y versátil ya que simplemente debemos indicar con un "`_`" dónde va a aparecer el argumento, p.ej., se pueden definir símbolos tan versatiles como

```
op if_then_else_fi : Bool Exp Exp -> Exp
op for(_;_;_) {_} : Nat Bool Bool Exp -> Exp
```

En concreto en el ejemplo `VENDING-MACHINE` tenemos el símbolo

```
op __ : State State -> State
```

que denota que el carácter "vacío" es un símbolo válido usado para concatenar estados. Y en el ejemplo `FACT` tenemos

```
op _! : Int -> Int .
```

que denota el símbolo factorial en notación postfija.

2.5.1. Tipos de datos ordenados y sobrecarga de operadores

En Maude se pueden crear tipos de datos ordenados o divididos en jerarquías. Por ejemplo, podemos indicar que los números naturales se dividen en números naturales positivos y el cero usando la palabra reservada `subsort` de la siguiente forma:

```
sorts Nat Zero NzNat .
subsort Zero < Nat .
subsort NzNat < Nat .
op 0 : -> Zero .
op s : Nat -> NzNat .
```

De esta forma, la expresión `s(0)` es de tipo `NzNat` y a la vez es de tipo `Nat`, mientras que no es de tipo `Zero`. E igualmente, la expresión `0` es de tipo `Zero` y `Nat`, pero no es de tipo `NzNat`.

Otra característica interesante del sistema de tipos de Maude es la sobrecarga de operadores. Por ejemplo, podemos reutilizar el símbolo `0` en el tipo de datos `Binary` sin ningún problema:

```
sorts Nat Zero NzNat .
subsort Zero < Nat .
subsort NzNat < Nat .
op 0 : -> Zero .
op s : Nat -> NzNat .

sort Binary .
op 0 : -> Binary .
op 1 : -> Binary .
```

En este caso, pueden surgir ambigüedades sobre algún término que se resuelven especificando el tipo detrás del término, por ejemplo `(0).Zero` ó `(0).Binary`. El sistema informará sólo de ambigüedades que no pueda resolver por su cuenta.

La unión de la sobrecarga de símbolos y los tipos ordenados le confiere una gran flexibilidad al lenguaje. Por ejemplo, podemos redefinir el anterior tipo de datos de lista de números naturales de la siguiente forma, donde `ENatList` denota lista vacía (es decir `nil`) y `NeNatList` denota lista no vacía de elementos:

```
sorts NatList ENatList NeNatList .
subsort ENatList < NatList .
op nil : -> ENatList .
subsort NeNatList < NatList .
op _:_ : Nat NeNatList -> NeNatList .
op _:_ : Nat ENatList -> NeNatList .
```

2.5.2. Propiedades avanzadas (o algebraicas) de los símbolos

El lenguaje Maude incorpora la posibilidad de especificar símbolos con propiedades algebraicas extra como asociatividad, conmutatividad, elemento neutro, etc. que facilitan mucho la creación de programas.

$$Asociatividad\ X + (Y + Z) = (X + Y) + Z$$

$$Conmutatuvidad\ X + Y = Y + X$$

$$Elemento\ neutro\ X + 0 = X$$

De hecho, permiten definir estructuras de datos basadas en listas y conjuntos de forma muy sencilla e integrada dentro del lenguaje, mejorando el rendimiento de la ejecución y la compresión del programa o modelo. Por ejemplo, podemos redefinir el tipo de datos lista de números naturales de la siguiente forma:

```
sorts NatList ENatList NeNatList .
subsort ENatList < NatList .
op nil : -> ENatList .
subsort Nat < NeNatList < NatList .
op _:_ : NatList NatList -> NeNatList [assoc] .
```

donde `_:_` es un símbolo asociativo, es decir no hacen falta los paréntesis para separar los términos. Nótese que los dos argumentos del símbolo `_:_` tienen que ser del mismo tipo para poder indicar que el símbolo es asociativo. Ahora Maude entiende que las siguientes expresiones significan exactamente lo mismo

```
s(0) : s(s(0)) : nil
s(0) : (s(s(0)) : nil)
(s(0) : s(s(0))) : nil
```

Otra posibilidad es añadir un elemento neutro al operador asociativo:

```
sorts NatList .
subsort Nat < NatList .
op nil : -> NatList .
op _:_ : NatList NatList -> NatList [assoc id: nil] .
```

donde ahora `_:_` es un símbolo asociativo y el término `nil` es el elemento neutro del tipo de datos, que por lo tanto se puede eliminar salvo cuando aparece solo. Ahora Maude entiende que las siguientes expresiones significan exactamente lo mismo

```
s(0) : s(s(0)) : nil
s(0) : s(s(0))
nil : s(0) : nil : s(s(0)) : nil
```

Y podemos añadir la propiedad de conmutatividad a la lista, creando el tipo de datos multiconjunto:

```
sorts NatMultiSet .
subsort Nat < NatMultiSet .
op nil : -> NatMultiSet .
op _:_ : NatMultiSet NatMultiSet -> NatMultiSet [assoc comm id: nil] .
```

donde la propiedad de conmutatividad indica que se puede intercambiar el orden de los elementos. Ahora Maude entiende que las siguientes expresiones significan exactamente lo mismo

```
0 : s(0) : s(s(0)) : s(0)
0 : s(0) : s(0) : s(s(0)) : nil
s(0) : 0 : s(s(0)) : s(0) : nil
nil : s(0) : nil : s(s(0)) : nil : s(0) : nil : 0 : nil
```

Y finalmente podemos añadir la propiedad de que no pueden haber elementos iguales, convirtiendo el multiconjunto en un conjunto:

```
sorts NatSet .
subsort Nat < NatSet .
op nil : -> NatSet .
op _:_ : NatSet NatSet -> NatSet [assoc comm id: nil] .
eq X:Nat : X:Nat = X:Nat .
```

donde la ecuación elimina aquellas ocurrencias repetidas de un término. Ahora Maude entiende que las siguientes expresiones significan exactamente lo mismo

```
0 : s(0) : s(s(0))
0 : s(0) : s(0) : s(0) : s(0) : s(s(0)) : nil
s(0) : 0 : s(s(0)) : s(0) : nil
nil : s(0) : nil : s(s(0)) : nil : s(0) : nil : 0 : nil
```

2.6. Declaración de Funciones

Aquellos operadores o símbolos que dispongan de reglas o ecuaciones que los definan son denominados *funciones* mientras que los que no dispongan de reglas o ecuaciones son denominados *constructores*. Nótese que es obligatorio en Maude declarar el tipo de todas las funciones, el tipo de todas las variables, etc. En otros lenguajes funcionales, como Haskell, esto no es necesario aunque se recomienda. En particular, esto puede ayudar a detectar fácilmente errores en el programa, cuando se definen funciones que no se ajustan al tipo declarado.

Respecto a las reglas/ecuaciones que definen las funciones, éstas pueden ser de la forma

`rl` $f(t_1, \ldots, t_n)$ `=>` e .
`eq` $f(t_1, \ldots, t_n)$ `=` e .

donde $\texttt{t}_1, \ldots, \texttt{t}_\texttt{n}$ y e son términos. Las ecuaciones pueden etiquetarse con la palabra reservada `owise` (otherwise) y en ese caso se indica que sólo se aplicará si ninguna otra ecuación para ese símbolo es aplicable. **Cuidado** porque el `owise` sólo se puede aplicar a una ecuación, nunca a una regla ya que tienen un significado indeterminista. Por ejemplo, podemos escribir

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

Las funciones se pueden definir también mediante reglas/ ecuaciones condicionales

`crl` $f(t_1, \ldots, t_n)$ `=>` e `if` c .
`ceq` $f(t_1, \ldots, t_n)$ `=` e `if` c .

donde la condición `c` es un conjunto de emparejamientos de la forma `t := t'` separados por el operador `/\`. Un emparejamiento `t := t'` indica que el término `t'` debe tener la forma del término `t`, instanciando las variables de `t` si es necesario, ya que las variables de `t` pueden ser usadas en la expresión `e` de la regla para extraer información de `t'`. Las ecuaciones condicionales sólo pueden aplicarse si la condición tiene éxito. También es posible definir una ecuación condicional en la que las guardas sean expresiones de tipo `Bool` en vez de `t := t'`; en ese caso se interpretan como `true := t`. Por ejemplo, podemos escribir la anterior función factorial de la siguiente forma:

```
ceq N ! = 1 if N == 0 .
ceq N ! = (N - 1)! * N if N =/= 0 .
```

donde la igualdad '==' se evalúa a `true` si ambas expresiones son iguales y '=/=' se evalúa a `true` si ambas expresiones son distintas. Nótese que en este caso, se puede usar también el operador condicional `if_then_else_fi`:

```
eq N ! = if N == 0 then 1 else (N - 1)! * N fi .
```

Además, debido a los tipos de datos ordenados, se permiten expresiones lógicas de la forma `t :: T` que se evalúan a `true` si la expresión `t` es del tipo `T`. Por ejemplo, esto puede ser útil en ecuaciones condicionales como:

```
op emptyList : NatList -> Bool .
eq emptyList(NL) = NL :: ENatList .
```

donde se dice que una lista `NL` está vacía, es decir `emptyList` retorna `true`, si `NL` es del tipo `ENatList`.

2.6.1. Búsqueda eficiente de elementos en listas y conjuntos

Una ventaja de disponer de listas, multiconjuntos y conjuntos es que determinadas operaciones de búsqueda y emparejamiento de patrones resultan mucho más rápidas. De hecho, más rápidas que en otros lenguajes declarativos como Prolog o Haskell. Por ejemplo, la pertenencia de un elemento a una lista se realiza de forma secuencial en muchos lenguajes declarativos:

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

op _in_ : Nat NatList -> Bool .
eq N:Nat in nil
 = false .
eq N:Nat in (X:Nat : XS:NatList)
 = (N:Nat == X:Nat) or-else (N:Nat in XS:NatList) .
```

Sin embargo, cuando disponemos de un operador asociativo con un elemento neutro, esta operación se hace de forma mucho más elegante y eficiente como sigue a continuación:

```
sort NatList .
subsort Nat < NatList .
op nil : -> NatList .
op _:_ : NatList NatList -> NatList [assoc id: nil] .

op _in_ : Nat NatList -> Bool .
eq N:Nat in (L1:NatList : N:Nat : L2:NatList)
 = true .
eq N:Nat in L:NatList
 = false [owise] .
```

La expresión "`1 in 1 : 2 : 3`" se puede ver como "`1 in nil : 1 : 2 : 3`" gracias a la propiedad del elemento neutro, donde "`L1:NatList`" se emparejaría con "`nil`", "`N:Nat`" con "`1`" y "`L2:NatList`" con "`2 : 3`"; ocurre algo parecido con "`2 in 1 : 2 : 3`" y "`3 in 1 : 2 : 3`". Esto tiene la ventaja de que es el propio sistema el que decide la mejor técnica de búsqueda y así no está restringido al mecanismo usado por el programador. Además, en el caso de un conjunto (o multiconjunto) es aún más simple gracias a la conmutatividad:

```
sort NatSet .
subsort Nat < NatSet .
op empty : -> NatSet .
op _:_ : NatSet NatSet -> NatSet [assoc comm id: nil] .

op _in_ : Nat NatSet -> Bool .
eq N:Nat in (N:Nat : L:NatSet)
 = true .
eq N:Nat in L:NatSet
 = false [owise] .
```

2.7. 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://moment.dsic.upv.es`. 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 > .`".

`reduce 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
```

Y si especificamos una función con un comportamiento indeterminista, por ejemplo la función `_+-` que o bien incrementa o decrementa un número dado:

```
mod INC-DEC is
  protecting INT .
  op _+- : Int -> Int .
  var N : Int .
  rl N +- => N + 1 .
  rl N +- => N - 1 .
endm
```

y asumiendo que está almacenada en el fichero `masmenos.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 masmenos.maude
Maude> rew 1 +- .
rewrite in INC-DEC : 1 +- .
rewrites: 2 in 0ms cpu (0ms real) (71428 rewrites/second)
result NzNat: 2
Maude> quit
```

pero siempre va a devolver el valor `2` y si queremos buscar todas las soluciones usaremos el comando `search`:

```
$ 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 ejemplo3.maude
Maude> search 1 +- =>! X:Int .
search in INC-DEC : 1 +- =>! X:Int .

Solution 1 (state 1)
states: 3  rewrites: 4 in 0ms cpu (0ms real) (102564 rewrites/second)
X:Int --> 2

Solution 2 (state 2)
states: 3  rewrites: 4 in 0ms cpu (0ms real) (57971 rewrites/second)
X:Int --> 0

No more solutions.
states: 3  rewrites: 4 in 0ms cpu (0ms real) (44444 rewrites/second)
```

Es importante darse cuenta que en el módulo anterior no se puede ejecutar el comando `red 1 +-` porque está definido usando reglas y no ecuaciones

```
Maude> load masmenos.maude
Maude> red 1 +- .
reduce in INC-DEC : 1 +- .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result Int: 1 +-
Maude> quit
```

3. Objetivo de la práctica

El objetivo de esta práctica consiste en responder a la serie de preguntas cortas que se muestran a continuación, modificando los programas de la forma requerida.

Evaluación: La evaluación de esta práctica se realizará subiendo las respuestas a la tarea de PoliformaT.

3.1. Batería de celdas

El siguiente módulo funcional modela una batería de N celdas (con $N \geq 0$) como una combinación de los símbolos `o`, `+` y `-` usando el estilo de programación de Haskell. Una lista se representa con la estructura `cabeza ^ cola`, donde el símbolo `^` lo escoge el programador a diferencia de Haskell, que solo permite el símbolo `_:_`. Por ejemplo, para el caso de una batería de 2 celdas las posibles combinaciones se representan como: `o ^ o ^ nil` (batería llena), `+ ^ + ^ nil`, `+ ^ o ^ nil`, `o ^ + ^ nil`, `- ^ o ^ nil`, `- ^ + ^ nil`, `o ^ - ^ nil`, `+ ^ - ^ nil`, `- ^ - ^ nil` (batería vacía).

```
fmod BATTERY-HASKELL is
  sorts Cell Battery .

  op o : -> Cell  .
  op + : -> Cell  .
  op - : -> Cell  .
  op nil : -> Battery  .
  op _^_ : Cell Battery -> Battery  .
```

```
  op consume : Battery -> Battery .

  var Bt : Battery .

  eq consume(o ^ Bt) = + ^ Bt .
  eq consume(+ ^ Bt) = - ^ Bt .
  eq consume(- ^ Bt) = - ^ consume(Bt) .
endfm
```

Una ejecución del módulo puede ser:

```
Maude> load battery-haskell.maude
Maude> reduce in BATTERY-HASKELL : consume(- ^ o ^ nil) .
rewrites: 2 in 0ms cpu (0ms real) (2000000 rewrites/second)
result Battery: - ^ + ^ nil
```

Basándonos en este módulo y usando tus propias palabras responde a las siguientes cuestiones:

> (Pregunta 1) ¿Qué se obtiene tras ejecutar el siguiente comando y cómo se podría arreglar?
>
> ```
> Maude> red in BATTERY-HASKELL : consume(- ^ - ^ nil) .
> ```

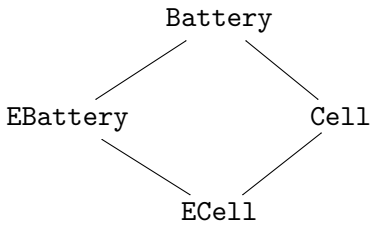
3.2. Batería de celdas mejorada usando subtipos y axiomas en Maude

Maude permite definir una jerarquía de tipos para añadir más información a nuestros datos. Por ejemplo, usando la relación de subtipado (`subsort`), podemos definir el siguiente módulo funcional que nos permite distinguir, dentro de los tipos `Cell` y `Battery`, cuándo una celda o una batería está vacía (que modelamos mediante los tipos de datos `ECell` y `EBattery`). Además indicamos que las baterías son listas de celdas usando la asociatividad y el elemento identidad `nil`.

```
fmod BATTERY-MAUDE is
  sorts Cell ECell Battery EBattery .

  subsorts ECell < Cell < Battery .
  subsorts ECell < EBattery < Battery .

  op o : -> Cell  .
  op + : -> Cell  .
  op - : -> ECell  .
  op nil : -> EBattery  .
  op _^_ : EBattery EBattery -> EBattery [assoc id: nil] .
  op _^_ : Battery Battery -> Battery [assoc id: nil] .
  op consume : Battery -> Battery .

  var Bt : Battery . var EBt : EBattery .

  eq consume(EBt ^ o ^ Bt) = EBt ^ + ^ Bt .
  eq consume(EBt ^ + ^ Bt) = EBt ^ - ^ Bt .
  eq consume(Bt) = Bt [owise] .
endfm
```

Basándonos en este módulo y usando tus propias palabras responde a las siguientes cuestiones:

(Pregunta 2) Gráficamente, se puede representar el orden parcial de la relación de subtipado como un grafo:



Muestra un ejemplo de un término que sea de tipo `EBattery` pero no `ECell` y un término que sea de tipo `Battery` pero no `EBattery` ni `Cell`.

(Pregunta 3) ¿Para qué nos puede servir diferenciar dentro del tipo `Cell` el subtipo `ECell`?

(Pregunta 4) Para una batería de 2 celdas, muestra el conjunto de valores que pueden tomar una variable de tipo `EBattery` y una variable de tipo `Battery`.

(Pregunta 5) ¿Qué tipo se obtiene tras ejecutar el siguiente comando y qué significa?

```
Maude> red in BATTERY-MAUDE : consume(- ^ -) .
```

3.3. Batería de celdas mejorada usando indeterminismo en Maude

Partiendo del módulo `BATTERY-MAUDE` del ejercicio anterior se añade el siguiente módulo de reescritura que consume aleatoriamente una celda en un extremo de la batería (izquierdo o derecho):

```
mod BATTERY-LEFT-RIGHT is
  protecting BATTERY-MAUDE .

  op consume-left-right : Battery -> Battery .

  var EBt : EBattery .
  var Bt : Battery .

  rl consume-left-right(EBt ^ o ^ Bt)  => EBt ^ + ^ Bt .
  rl consume-left-right(EBt ^ + ^ Bt)  => EBt ^ - ^ Bt .
  rl consume-left-right(Bt  ^ o ^ EBt) => Bt ^ + ^ EBt .
  rl consume-left-right(Bt  ^ + ^ EBt) => Bt ^ - ^ EBt .
  eq consume-left-right(EBt) = EBt .
endm
```

(Pregunta 6) Usando tus propias palabras, ¿qué se obtiene cuando se ejecuta el siguiente comando y por qué?

```
Maude> red in BATTERY-LEFT-RIGHT : consume-left-right(- ^ o ^ o) .
```

(Pregunta 7) ¿Y con el siguiente comando?

```
Maude> rew in BATTERY-LEFT-RIGHT : consume-left-right(- ^ o ^ o) .
```

(Pregunta 8) ¿Y con el siguiente comando?

```
Maude> search in BATTERY-LEFT-RIGHT : consume-left-right(- ^ o ^ o) =>! Bt:Battery .
```

(Pregunta 9) ¿Y con el siguiente comando?

```
Maude> search in BATTERY-LEFT-RIGHT : consume-left-right(- ^ o ^ o) =>* Bt:Battery .
```

3.4. Ejercicio libre sobre la batería de celdas

Escribe un programa que consuma una celda de la batería, pero que sea cualquiera de las disponibles, en vez de consumir solamente la que esté en uno de los extremos