MODELOS FORMALES DE COMPUTACIÓN

Semántica de continuaciones para Programación Funcional

---

1. ¿Cómo usar la semántica de continuaciones?

En PoliformaT tenéis disponibles distintos ficheros que corresponden a las semánticas de continuaciones descritas en teoría. Cada una de estas semánticas se han implementado como un programa en el lenguaje de programación Maude.

Por ejemplo, el tema 4 de programación funcional tiene asociado un fichero "Tema 4.maude" donde se incluye la implementación en Maude de la semántica descrita en teoría.

1.1. Estructura de cada fichero de Maude

Cada fichero en Maude que describe una semántica de continuaciones tiene una estructura muy simple:

- Hay diversos módulos funcionales de Maude, descritos como "`fmod Nombre is … endfm`". Pero hay un primer grupo de módulos funcionales que definen la sintaxis y un segundo grupo que define la semántica.

- La gramática BNF vista en teoría para cada lenguaje se codifica como un conjunto de definiciones de símbolos en Maude. Por ejemplo, la definición del operador de asignación en la gramática BNF

```
Exp ::= Name := Exp
```

donde `Exp` y `Name` son símbolos no terminales de la gramática y ":=" es un símbolo terminal (una palabra reservada) da lugar a la siguiente definición en Maude

```
op _:=_ : Name Exp -> Exp .
```

donde `Exp` y `Name` son tipos de datos definidos al principio del fichero con la palabra `sort` y "`_:=_`" es un símbolo en Maude.

- Cada una de las definiciones de la semántica por continuaciones se codifica también en Maude usando ecuaciones. Por ejemplo, la definición semántica de qué hacer cuando nos encontramos un número entero, es decir,

$$\text{exp}(I,Env) \curvearrowright K \rightleftharpoons \text{val}(I) \curvearrowright K$$

se codifica en Maude como una ecuación

```
eq k(exp(I, Env) -> K) = k(val(int(I)) -> K) .
```

con la pequeña diferencia de incluir el símbolo k que identifica la componente asociada a la pila de continuaciones.

1.2. Ejecutar una semántica descrita en Maude

No es necesario ser un experto programador en Maude para ser capaz de entender cómo se describe una semántica en Maude y cómo podemos escribir programas aceptados por esa semántica y ejecutarlos.

Cada fichero de Maude asociado a un tema de la asignatura incluye varios programas de ejemplo. La forma de cargar uno de esos ficheros en el entorno Maude sería

```
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
           Maude 2.7.1 built: Jun 27 2016 16:43:23
            Copyright 1997-2016 SRI International
                  Mon Oct 29 17:27:56 2018
Maude> load /usr/usuario/MFC/Tema\ 4.maude
.........
```

mostrando por pantalla la ejecución de todos los programas de ejemplo que incluye. Se puede cambiar de directorio con el comando "`cd`" y se puede conocer el directorio actual con el comando "`pwd`". Si haces las prácticas con Maude para Windows, las rutas de directorio son de la forma "`/cygwin/c/Users/usuario`".

El programa Maude asociado a la semántica de continuaciones del Tema 4 incluye una función `eval` donde se le pasa justamente el programa que deseamos ejecutar. Esta función `eval` se encarga de generar una configuración inicial del lenguaje con todos los componentes inicializados de forma apropiada y con el programa a ejecutar situado en la cabeza de la pila de continuaciones. Además esta función `eval` nos devuelve el valor resultado de la ejecución del programa, ya que es una expresión funcional.

Por ejemplo, dado una simple función f que simplemente suma los dos argumentos que recibe como parámetros, la codificación y ejecución usando la semántica de continuaciones del Tema 4 descrita en Maude sería la siguiente:

```
Maude> reduce in FUN-TEST : eval(let f(x,y) = x + y in f(1,2)) .
result Value: int(3)
```

Si en vez de considerar la semántica del Tema 4, consideramos la semántica del Tema 3, asociado a un lenguaje imperativo, la codificación de dicha función f y su ejecución serían distintas, ya que la función `eval` en ese tema necesita de una lista de datos de entrada.

```
Maude> reduce in SIMPLE-SEMANTICS :
    eval function f(x,y) {return(x + y)}
         function main () {print (f (1,2))}
    | nil .
result Value: int(3)
```

1.3. Tipos de datos

Las semánticas por continuaciones descritas en esta asignatura no realizan ninguna comprobación de tipos, por lo que hay que tener mucho cuidado al escribir los programas,

ya que no vamos a obtener ninguna información de error.

Por ejemplo, si nos equivocamos al introducir la función f de arriba y reemplazamos el símbolo de suma por la palabra `and`, Maude ejecutará la expresión `eval` introducida, devolviendo una configuración inválida (o incorrecta) de la semántica.

```
Maude> reduce in FUN-TEST : eval(let f(x,y) = (x and y) in f (1,2)) .
result [ValueList]: [nextLoc(3) mem([loc(0),closure(x,y, x and y, empty)]
[loc(1),int(1)] [loc(2),int(2)]) k(val (int(1),int(2)) -> and -> stop)]
```

1.4. Notación

Cada semántica viene predefinida con identificadores de una única letra. Si queréis escribir un identificar (nombre de función o variable o parámetro) de más de una letra, p.ej. máximo ó fact, hay que utilizar el símbolo del acento que se encuentra al lado de la tecla del cero en el teclado español, es decir, 'maximo ó 'fact.

Puede ser más fácil escribir las funciones en notación infija con paréntesis "f(x,y)" en vez de "(f x y)" aunque acepta ambas.

Si se define una función recursiva, tendrás que utilizar una expresión del tipo "let rec" en vez de "let". Las operaciones para manejo de listas son "`null?`", "`car`" y "`cdr`". Así que no podrás escribir "`l == []`" porque da error, hay que escribir "`null?(l)`".

2. Ejercicios básicos a resolver

2.1. Escribir una función recursiva que devuelva el sumatorio desde un valor entero hasta otro.

2.2. Define una función binaria que devuelva el mayor de sus dos argumentos.

2.3. Define una función que calcule el factorial de un número.

3. Ejercicios a resolver sobre tipos listas

3.1. Define una función para contar cuántas veces aparece un elemento en una lista.

3.2. Define una función que reemplace en una lista todas las ocurrencias de un elemento `n` por otro elemento `p`.

3.3. Define una función que extraiga de una lista los números menores que otro número dado (inténtala usando orden superior).

4. ¿Cómo extender una semántica de continuaciones?

Como ya hemos comentado durante la asignatura, hemos considerado simplemente un paso de parámetros por valor en todas las semánticas estudiadas, para simplificar la exposición. Sin embargo, es muy fácil extender cualquiera de las semánticas para tener paso de parámetros por referencia usando el mismo concepto de manipulación de punteros con & y * que usa el lenguaje C.

Es decir, quisiéramos ser capaces de ejecutar el siguiente programa

```
let z = 0 and f(x,y,z) = { * z := x + y; * z } in f(1,2,& z)
```

Para modificar la semántica de forma apropiada, tienes que añadir dos módulos, `POINTER-SYNTAX` para la sintaxis y `POINTER-SEMANTICS` para la semántica, modificar la definición sintáctica del símbolo "`_:=_`" y enlazar los dos módulos nuevos de forma apropiada:

```
fmod POINTER-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  ops &_ *_ : Name -> Exp .
endfm

fmod ASSIGNMENT-SYNTAX is
  including GENERIC-EXP-SYNTAX .
  op _:=_ : Exp Exp -> Exp .
endfm

fmod POINTER-SEMANTICS is
  including POINTER-SYNTAX .
  including ASSIGNMENT-SYNTAX .
  including FUN-HELPING-OPERATIONS .
  op pointer : Location -> Value .
  var X : Name .  var L1 L2 : Location .  var V : Value .
  var K : Continuation .  var Env : Env .  var Mem : Store . var E : Exp .

  eq k(exp(& X, Env) -> K) = k(val(pointer(Env[X])) -> K) .
  eq k(exp(* X, ([X,L1] Env)) -> K) mem([L1,pointer(L2)] [L2,V] Mem)
   = k(val(V) -> K) mem([L1,pointer(L2)] [L2,V] Mem) .
  eq k(exp(* X := E, ([X,L1] Env)) -> K) mem([L1,pointer(L2)] Mem)
   = k(exp(E, ([X,L1] Env)) -> writeTo(L2) -> val(nothing) -> K) mem([L1,pointer(L2)] Mem) .
endfm

fmod FUN-SYNTAX is
  ...
  including POINTER-SYNTAX .
  ...
endfm

fmod FUN-SEMANTICS is
  ...
  including POINTER-SEMANTICS .
  ...
endfm
```

Y deberías ser capaz de ejecutar el programa anterior

```
reduce in FUN-TEST : eval(let z = 0 and f (x,y,z) = {* z := x + y ; * z} in f (1,2,& z)) .
result Value: int(3)
```

5. Ejercicios a resolver sobre paso por referencia

5.1. Define una función binaria que intercambie los valores de sus dos argumentos.

5.2. Define una función que extraiga el valor mínimo y máximo de una lista de enteros usando paso por referencia.