MODELOS FORMALES DE COMPUTACIÓN

Semántica de continuaciones para Programación Concurrente

---

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 7 de programación concurrente con hilos de ejecución tiene asociado un fichero "Tema 7.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

```
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,

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

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.

Por ejemplo, el programa Maude asociado a la semántica de continuaciones del Tema 7 incluye una función `eval` donde se le pasa justamente el programa que deseamos ejecutar y devuelve el último valor de la ejecución. 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.

Sin embargo, la semántica de hilos de ejecución incorpora indeterminismo y éste se codifica en Maude con reglas, mientras que las partes deterministas se codifican con ecuaciones. Por lo tanto, ya no se puede utilizar el comando `reduce` de Maude, ya que solo funciona cuando toda la semántica es determinista. Cuando hay indeterminismo Maude proporciona dos comandos: `rewrite` y `search` dependiendo, respectivamente, de si estamos interesados en obtener un único camino o explorar todos los caminos posibles,

Por ejemplo, dado el siguiente ejemplo de un programa que genera dos hilos de ejecución con una asignación cada uno, aparte del hilo principal, la codificación y ejecución usando cada uno de los dos comandos sería la siguiente:

```
Maude> rewrite eval (let x = 0 in {spawn(x := x + 1) ; spawn(x := x + 1) ; x}) .
result Value: int(0)

Maude> search eval (let x = 0 in {spawn(x := x + 1) ; spawn(x := x + 1) ; x}) =>!
X:Value .

Solution 1 (state 4)
X:Value --> int(0)

Solution 2 (state 15)
X:Value --> int(1)

Solution 3 (state 25)
X:Value --> int(2)

No more solutions.
```

Hay que tener cuidado con un programa que no termine, ya que un comando rewrite o search puede que no termine o genere un número infinito de soluciones. En tal caso, el comando tiene un campo para introducir un límite de soluciones, p.ej.

```
search [3] eval (…) =>! X:Value .
```

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.

2. Ejercicios a resolver

2.1. ¿Qué ocurre en la ejecución del siguiente programa?

```
let x = 0
in {
  spawn(x := x + 1) ;
  spawn(x := x + 1) ;
  x
}
```

2.2. ¿Qué ocurre en la ejecución del siguiente programa?

```
let x = 0
in {
  spawn(acquire lock(1) ; x := x + 1 ; release lock(1)) ;
  spawn(acquire lock(1) ; x := x + 1 ; release lock(1)) ;
  x
}
```

2.3. ¿Qué ocurre en la ejecución del siguiente programa?

```
let rec f(x,y) = spawn(f(x,y) ; x := x + 1 ; y := 1)
in
 let x = 0 and a = 0
 in {
     f(x,a) ;
     while (a == 0) {} ;
     x
 }
```

2.4. ¿Qué ocurre en la ejecución del siguiente programa?

```
let rec f(x,y) = { f(x,y) ; x := x + 1 ; y := 1 }
in
 let x = 0 and a = 0
 in {
     f(x,a) ;
     while (a == 0) {} ;
     x
 }
```

2.5. ¿Qué ocurre en la ejecución del siguiente programa?

```
let rec f(x,y) = spawn( x := x + 1 ; y := 1 )
in
 let x = 0 and a = 0
 in {
     f(x,a) ;
     while (a == 0) {} ;
     x
 }
```

2.6. ¿Qué ocurre en la ejecución del siguiente programa?

```
let rec f(x,y) = { x := x + 1 ; y := 1 }
in
 let x = 0 and a = 0
 in {
     f(x,a) ;
     while (a == 0) {} ;
     x
 }
```

2.7. ¿Qué ocurre en la ejecución del siguiente programa?

```
let x = 0 and a = 0
and f(x,y) = spawn(x := x + 1 ; y := 1)
in {
    f(x,a) ;
    while (a == 0) {} ;
    x
}
```

2.8. ¿Qué ocurre en la ejecución del siguiente programa?

```
let rec x = 0 and a = 0
and f() = spawn( x := x + 1 ; a := 1 )
in {
    f() ;
    while (a == 0) {} ;
    x
}
```

2.9. ¿Qué ocurre en la ejecución del siguiente programa?

```
let rec x = 0 and a = 0
and f() = { x := x + 1 ; a := 1 }
in {
    f() ;
    while (a == 0) {} ;
    x
}
```

2.10. ¿Qué ocurre en la ejecución del siguiente programa?

```
let rec x = 0 and a = 0
and f() = spawn( f(); x := x + 1 ; a := 1 )
in {
    f() ;
    while (a == 0) {} ;
    x
}
```