Fundamentos de la Computación
Trabajo de Laboratorio 5
Imbibición del lenguaje de Dijkstra en Haskell

16 de Junio de 2014

0. Preludio. En este laboratorio vamos a embeber en Haskell el lenguaje de Dijkstra[1], también llamado *Lenguaje de Comandos con Guardas*. La imbibición que definiremos es *superficial*, o sea, utilizaremos los tipos de Haskell para representar las expresiones del lenguaje embebido.

Comenzamos por darle nombre al módulo e ingresar los datos de los autores:

```
module Lab5 where
-- nombre y nro de estudiante 1
-- nombre y nro de estudiante 2
```

1. Tipos básicos

Las *variables* son Strings (listas de caracteres que se escriben entre " "):

```
type Var = String
```

Una *memoria* es una lista de variables con su valor asociado:

```
type Memoria = [(Var,Int)]
```

[1] *A Discipline of Programming.* Edsger W. Dijkstra, Prentice-Hall, 1976

?1. Definir las siguientes funciones que permitirán modificar y acceder a la Memoria:

a. Memoria vacia: es una memoria sin variables inicializadas

```
ini :: Memoria
ini = ...
```

b. Lookup: devuelve el valor de una variable en la Memoria (si la variable no fue inicializada, da error)

```
(@@) :: Var -> Memoria -> Int
(@@) = ...
```

c. Update: modifica el valor de una variable en la Memoria con un valor dado

```
upd ::  (Var,Int) -> Memoria -> Memoria
upd = ...
```

2. Expresiones. Las *expresiones* del lenguaje se construyen a partir de números enteros, variables y operadores aritméticos. Al contener variables, su valor depende del contenido de la Memoria, por lo que se representan como funciones del tipo `Memoria -> Int`.

```
type Exp = Memoria -> Int
```

Por ejemplo, la variable x se representa como la expresión:

```
\m -> "x" @@ m
```

la expresion $x * 2 + z$ se representa como la función:

```
\m -> ("x" @@ m) * 2 + ("z" @@ m)
```

y el valor de esta expresion en la memoria se obtiene aplicando directamente la expresion a la memoria. Por ejemplo:

```
(\m -> ("x" @@ m) * 2 + ("z" @@ m)) [("x",1),("z",5)] = 7.
```

Observar que las expresiones que son constantes también deben escribirse como funciones, o sea, la expresión 1 es en realidad la función `\m -> 1`.

3. Programas. Los *programas* son funciones que reciben una memoria y devuelven otra:

```
type Prog = Memoria -> Memoria
```

Las siguientes funciones definen los distintos tipos de programas del lenguaje de Dijkstra:

Asignación. Se asigna simultáneamente una lista de expresiones a una lista de variables:

```
as :: [(Var, Exp)] -> Prog
as = \ves -> \m -> updates (eval ves m) m
```

Esta función utiliza dos funciones auxiliares:

a. `eval:: [(Var,Exp)]-> Memoria -> [(Var,Int)]`, que recibe una lista de pares de variables y expresiones y una memoria, evalúa las expresiones en la memoria y devuelve una lista con los pares de variables y enteros obtenidos

b. `updates :: [(Var,Int)]-> Memoria -> Memoria` que recibe una lista de pares de variables y enteros y una memoria, y devuelve la memoria resultado de actualizar cada variable de la lista con el valor entero correspondiente.

?2. Definir funciones `eval` y `updates`.

Composición Secuencial. Es simplemente la ejecución de un programa y luego otro:

```
(>*>) :: Prog -> Prog -> Prog
(>*>) = \p1 -> \p2 -> \m -> p2 (p1 m)
```

Condicional con Guardas. Recibe una lista de pares de expresiones booleanas (guardas) y programas, y ejecuta alguno de los programas cuya guarda sea verdadera. Si ninguna guarda es verdadera da error.

Del mismo modo que las expresiones, las guardas son funciones:

```
type Guarda = Memoria -> Bool
```

Para definir la ejecución del if con guardas se utiliza una función auxi-liar `choice`, que recibe una lista de pares de guardas y programas y una memoria, y elije alguno de los programas para los cuales la guarda es verdadera. Como esta elección puede ser vacía, se utiliza un tipo llamado `Maybe`, que está definido del siguiente modo:

```
data Maybe a where {Nothing :: Maybe a ; Just :: a -> Maybe a}
```

Una expresión del tipo `Maybe a` puede ser de la forma `Nothing` (nada) o `Just x`, donde `x` es una expresión de tipo `a`.

La función `choice` tendrá entonces el tipo

```
choice :: [(Guarda,Prog)] -> Memoria -> Maybe Prog
```

y la función que implementa el if con guardas será la siguiente:

```
gif :: [(Guarda,Prog)] -> Prog
gif = \gps \m -> case choice gps m of {
                 Nothing -> error "gif: ninguna guarda verdadera";
                 Just p -> p m}
```

?3. Definir la función `choice`.

Iteración con guardas. Al igual que el if, el do con guardas recibe una lista de pares de guardas y programas, y ejecuta un programa cuya guarda sea verdadera. Este procedimiento se repite hasta que ninguna guarda sea verdadera, en cuyo caso el programa termina sin fallar.

```
gdo :: [(Guarda,Prog)] -> Prog
gdo = ...
```

?4. Definir la función `gdo`.

4. Ejemplos de programas. Algunos ejemplos de programas del lenguaje de Dijkstra embebidos en Haskell son los siguientes.

Swap. Este programa da vuelta dos variables. Observar que la asignación simultánea hace que no sea necesario utilizar una variable auxiliar:

```
swap :: Prog
swap = as [("x", \m-> "y"@@m), ("y", \m ->"x"@@m)]
```

Mínimo. Este programa calcula el mínimo de dos números y deja el resultado en la variable `"min"`:

```
minimo :: Prog
minimo = gif [(\m -> (x@@m)<(y@@m), as [("min", \m -> x@@m)] ),
              (\m -> (x@@m)==(y@@m), as [("min", \m -> x@@m)] ),
              (\m -> (x@@m)>(y@@m), as [("min", \m -> y@@m)] )]
```

Potencia. Finalmente, este programa calcula el resultado de elevar un número `"x"` a una potencia `"y"`, dejando el resultado en la variable `"p"`:

```
potencia :: Prog
potencia =  as [(p, \m -> 1)] >*>  -- p:=1
            gdo [(\m ->(y@@m>0),   -- guarda y>0
            as [(p,\m ->(p@@m)*(x@@m)),(y,\m ->(y@@m)-1)])]
                                                  -- p:=x*m, y:= y-1
```

?5. Programar en el lenguaje de guardas `fact :: Prog` que calcula el factorial de un número positivo guardado en una variablle `"n"`, dejando su resultado en una variable `"fact"`.

?6. Programar en el lenguaje de guardas `fib :: Prog` que calcula el n-ésimo número de fibonacci, donde n está guardado en una variablle `"n"`, y deja su resultado en una variable `"fib"`.

5. Entregables

Para obtener los 5 puntos de esta evaluación se pide programar las funciones indicadas anteriormente. En Aulas se encuentra publicado un archivo Haskell con todas las definiciones de que están en este repartido, y las funciones faltantes definidas como `undefined`. También se publicarán en Aulas casos de prueba para testear las funciones definidas.
Se deberá subir ese archivo con las definiciones completas antes del **jueves 26/6 a las 20hs**. Subir un archivo por cada estudiante con los nombres de todos los miembros el grupo (máximo 2).
No habrá defensa oral de este Laboratorio, pero en el parcial se incuirá una pregunta sobre el mismo.