Práctica de Programación Funcional

Grado en Matemáticas e Informática

2º semestre, curso 2014–2015

ETSI Informáticos, UPM

14 de mayo de 2015

Algunas instrucciones

La puntuación de cada ejercicio se indica en su encabezado correspondiente. Puede realizarse la práctica en grupos de hasta 3 alumnos. La práctica resuelta debe consistir en un fichero con extensión `.hs` o `.lhs`. El nombre y el primer apellido de los alumnos tienen que formar parte del nombre del fichero, por ejemplo, `ElenaNito_MartínTero_ManuelaMacías.hs`. La práctica debe entregarse adjunta en un correo electrónico dirigido al coordinador de la asignatura con el asunto `FUNCIONAL` antes de la fecha límite de entrega acordada.

Objetivo

El objetivo de esta práctica y su código de apoyo es ilustrar las bondades del alto nivel de abstracción, genericidad, evaluación perezosa, tipos algebraicos, y seguridad del tipado estático fuerte de la programación funcional en Haskell. El medio es mostrar un caso de uso típico: sintaxis concreta y abstracta, evaluadores, compiladores y máquinas abstractas. Los alumnos podrán poner en práctica también sus conocimientos transversales de estructuras de datos (árboles binarios) y procesadores de lenguajes (gramáticas y máquinas a pila). Será necesario no solo implementar sino también entender código escrito en Haskell. También se contrastan métodos de aprendizaje y trabajo: (a) asistir y atender en clase y practicar, o por el contrario no hacerlo y (b) confiar en la habilidad para encontrar información compleja en internet y razonar sobre la misma, (c) confiar en la bondad de los compañeros que han seguido (a), o combinar (b) y (c). Buena suerte.

Introducción: The GuarriC Programming Language

El lenguaje de programación GuarriC es un subconjunto restringido del lenguaje de programación C. A continuación mostramos la gramática del lenguaje en formato EBNF:

Programa	→	Bloque
Bloque	→	Comando*
Comando	→	Var = Exp ;
	|	`if` (Exp) {Bloque} Else
	|	`while` (Exp) {Bloque}
	|	`print` (Exp) ;
Else	→	λ | {Bloque}
Exp	→	Var | Lit | Exp Op Exp
Op	→	+ | - | * | && | || | == | >

Los no-terminales Var y Lit estarían definidos por una gramática regular que define los elementos léxicos o «tokens» de variables (cadenas de caracteres) y literales enteros. Para simplificar, no imponemos ninguna restricción sobre ellos, cualquier cadena o entero se consideran válidos.

Como ejemplo, el siguiente programa muestra por pantalla el valor de factorial de 6:

```
n = 6;
r = 1;
while (n > 0) {
  r = r * n;
  n = n - 1;
}
print (r);
```

El siguiente programa muestra por pantalla los números pares desde 10 hasta 2 excepto en el caso de 4, en el que muestra -4.

```
n = 10;
while (n > 0) {
  if (n == 4) {
    print (-4);
  } else {
    print (n);
  }
  n = n - 2;
}
```

Comentamos en más detalle las características de GuarriC. Todas las variables son de tipo entero. Un valor igual a 0 indica «falso» y cualquier otro valor distinto de 0 indica «verdadero». Todas las variables son globales y debe tener un valor antes de usarse en una expresión. No hay funciones locales. Sólo hay cuatro comandos: asignación, condicionales «if-then-else» (con «else»

opcional), bucle `while` y comando `print` para mostrar el valor de una expresión en una línea de pantalla. Las expresiones de GuarriC consisten en variables, literales enteros, u operadores binarios aplicados a dos (sub)expresiones. Los operadores binarios son suma, resta, multiplicación, conjunción y disyunción lógica evaluadas en corto-circuito, operador de igualdad y operador de «mayor que». Las reglas de asociación y precedencia son las mismas que en el lenguaje C.

En el fichero con código de apoyo `GuarriC.lhs` se definen los tipos de datos algebraicos que permiten representar el árbol de sintaxis abstracta de un programa en GuarriC. En esta práctica los programas en GuarriC estarán escritos en dicha sintaxis abstracta y no será necesario disponer de un analizador sintáctico o «parser» que traduzca de sintaxis concreta (p.ej., el programa del factorial de 6 mostrado arriba) a sintaxis abstracta.

He aquí los tipos de datos incluidos en el fichero `GuarriC.lhs`:

```
> data Programa  = Prog Bloque
> type Bloque    = [Comando]
> data Comando   = Asig String Exp
>                | If Exp Bloque Bloque
>                | While Exp Bloque
>                | Print Exp
> data BTree a b = Leaf a | Node (BTree a b) b (BTree a b)
> type Exp       = BTree LitVar Op
> data LitVar    = Lint Integer | Var String
> data Op        = Add | Sub | Mul | And | Or | Eq | Gt
```

En palabras: un programa es un bloque, un bloque es una lista de comandos (que puede estar vacía), y los comandos son los anteriormente descritos. Para representar expresiones se utilizan árboles binarios con elementos diferentes en nodos hojas y nodos internos. En las hojas se pueden tener variables o literales. En los nodos internos se puede tener un operador. En `GuarriC.lhs` se incluyen ejemplos de expresiones en sintaxis abstracta:

```
> exps :: [Exp]
> exps =
>  [ Leaf (Lint 1)
>  , Leaf (Lint 0)
>  , Node (Leaf (Lint 0)) And (Leaf (Lint 1))
>  , Node (Leaf (Lint 1)) And (Leaf (Lint 1))
>  , Node (Leaf (Lint 0)) Or (Leaf (Lint 1))
>  , Node (Leaf (Lint 0)) Or (Leaf (Lint 0))
>  , Node (Leaf (Lint 8)) Add (Leaf (Lint 10))
>  , Node (Node (Leaf (Lint 1)) Mul (Leaf (Lint 3)))
>         Add
>         (Node (Leaf (Lint 4)) Mul (Leaf (Lint 9)))
>  , Node (Node (Leaf (Lint 1)) Eq  (Leaf (Lint 1)))
>         And
```

```
>         (Node (Leaf (Var "x")) Gt  (Leaf (Var "y")))
>  , Node (Node (Leaf (Lint 1)) Eq  (Leaf (Var "z")))
>         Or
>         (Node (Leaf (Lint 4)) Gt  (Leaf (Lint 2)))
>  ]
```

Para obtener la sintaxis concreta desde GHCi:

```
*GuarriC> mapM_ print exps
1
0
0 && 1
1 && 1
0 || 1
0 || 0
8 + 10
1 * 3 + 4 * 9
1 == 1 && x > y
1 == z || 4 > 2
```

En `GuarriC.lhs` también se ofrece un ejemplo de programa en sintaxis abstracta que imprime el valor de 1 + 1:

```
> muestra :: Programa
> muestra = Prog
>           [ Asig "n" (Node (Leaf (Lint 1)) Add (Leaf (Lint 1)))
>           , Print (Leaf (Var "n"))
>           ]
```

Para obtener su sintaxis concreta:

```
*GuarriC> print muestra
n = 1 + 1;
print (n);
```

El fichero `GuarriC.lhs` también ofrece las funciones para árboles binarios «fold» y `show`. Esta última serializa un árbol binario de expresiones a una cadena de carcateres para poder mostrarla por pantalla mediante el comando `print` de Haskell.

Ejercicio 1 (3 puntos)

Se Pide: Escribir en sintaxis abstracta de GuarriC los programas anteriormente mostrados. Deben llamarse `fact6` y `notFour`:

```
fact6 :: Programa
fact6 = ...   --completar--

notFour :: Programa
notFour = ... --completar--
```

Puede usarse la función `print` de Haskell sobre esos identificadores para comprobar la solución. Por ejemplo, `print fact6` debe mostrar el programa en sintaxis concreta mostrado en la introducción.

Ejercicio 2 (2 puntos)

Se Pide: Implementar la función

```
getVars :: Exp -> [String]
```

Que devuelve la lista de variables en inorden de la expresión (árbol binario) que toma como parámetro. Por ejemplo la expresión `map getVars exps` devuelve

```
[[],[],[],[],[],[],[],[],["x","y"],["z"]]
```

Ejercicio 2 (3 puntos)

Vamos a definir en Haskell un pequeño compilador para el lenguaje GuarriC como una función que toma un valor de tipo `Programa` y genera una lista de instrucciones para una arquitectura concreta. En este ejercicio vamos a centrarnos exclusivamente en la compilación de expresiones de GuarriC a instrucciones de la máquina a pila Matarile 69000, cuyo juego de instrucciones en sintaxis abstracta se especifica en el fichero `GuarriC.lhs`:

```
> data Instruccion = PUSH Integer
>                  | ADD
>                  | SUB
>                  | MUL
>                  | TEZ
>                  | TNZ
>                  | TGZ
>                    deriving Show
>
> type Codigo  = [Instruccion]
```

La máquina a pila toma una lista de instrucciones que evalúan de forma postfija una expresion usando una pila auxiliar de valores enteros. La instrucción `PUSH` introduce un valor entero en la pila. Las instrucciones `ADD`, `SUB` y `MUL` sacan de la pila dos valores *en el orden adecuado* y respectivamente los suman, restan, o multiplican dejando el resultado en la cima de la pila. Las instrucciones `TEZ`, `TNZ`, y `TGZ` sacan un valor entero *i* de la pila, e introducen en la pila el valor entero adecuado que representa el valor booleano que indica respectivamente si *i* es igual a cero, o distinto de cero, o mayor que cero.

La función `runExp` del fichero `GuarriC.lhs` implementa la máquina Maratile 69000 y da una pista de qué instrucciones debe generar el compilador de expresiones:

```
> type Stack = [Integer]
>
> push :: Integer -> Stack -> Stack
> push = (:)
>
> pop :: Stack -> (Integer,Stack)
> pop stk = (head stk, tail stk)

> runExp :: Stack -> Codigo -> Stack
> runExp stk c = foldl runInsts stk c
>   where
>   runInsts :: Stack -> Instruccion -> Stack
>   runInsts stk i = runIns i
>     where
>     (a, stk')  = pop stk
>     (b, stk'') = pop stk'
>     runIns :: Instruccion -> Stack
>     runIns (PUSH i) = push i stk
>     runIns ADD      = push (b + a) stk''
>     runIns SUB      = push (b - a) stk''
>     runIns MUL      = push (b * a) stk''
>     runIns TEZ      = push (fromIntegral $ fromEnum (a == 0)) stk'
>     runIns TNZ      = push (fromIntegral $ fromEnum (a /= 0)) stk'
>     runIns TGZ      = push (fromIntegral $ fromEnum (a > 0))  stk'
```

Para obtener el valor de una expresión con variables es necesario proporcionar el estado de las variables. Para representar el estado utilizaremos listas de tuplas donde el primer componente es la cadena de caracteres de la variable y el segundo componente es el valor que tiene esa variable:

```
> type Estado = [(String,Integer)]
```

Se Pide: Implementar la función:

```
compExp :: Estado -> Exp -> Codigo
```

Que toma una lista de tuplas variable-valor (un estado) y una expresión, y genera como resultado la lista de instrucciones del Matarile 69000 que evalúan la expresión. Para simplificar, en el caso de que la expresión contenga variables que no aparecen en el estado se debe producir un error en tiempo de ejecución.

Por ejemplo:

```
compExp [("z",1)]
        (Node (Node (Leaf (Lint 1)) Eq  (Leaf (Var "z")))
              Or
              (Node (Leaf (Lint 4)) Gt  (Leaf (Lint 2))))
```

debe devolver como resultado:

```
[PUSH 1,PUSH 1,SUB,TEZ,PUSH 4,PUSH 2,SUB,TGZ,ADD,TNZ]
```

En cambio,

```
compExp [("z",1)]
        (Node (Node (Leaf (Lint 1)) Eq  (Leaf (Var "z")))
              Or
              (Node (Leaf (Lint 4)) Gt  (Leaf (Lint 2))))
```

puede devolver algo parecido a ésto:

```
[PUSH 1,PUSH *** Exception: variable 'z' sin valor
```

La función `compExp` debe producir instrucciones de acuerdo al comportamiento de la función `runExp` que implementa la Matarile 69000. Es recomendable familiarizarse primero con el código de `runExp`.

Ejercicios 4 (1 punto)

En el fichero `GuarriC.lhs` se implementa también un intérprete de expresiones `evapExp`. Dicho intérprete toma un estado como primer parámetro, una expresión como segundo parámetro, y devuelve el valor entero resultado de evaluar la expresión. Por ejemplo:

```
evalExp [("z",1)]
        (Node (Node (Leaf (Lint 1)) Eq  (Leaf (Var "z")))
              Or
              (Node (Leaf (Lint 4)) Gt  (Leaf (Lint 2))))
```

devuelve 2, que es un valor mayor que cero

Para divertimento de los alumnos más motivados que han seguido la asignatura con celo y entusiasmo y que desean ver un ejemplo completo de código monádico con efectos, el fichero

`GuarriC.lhs` incluye también la función `evalP` que implementa un sencillo intérprete de todo GuarriC. Si lo desean, pueden usarlo para ejecutar sus programas GuarriC escritos en sintáxis abstracta. Por ejemplo, `evalP muestra` imprime por pantalla el valor 2, mientras que `evalP notFour` muestra la siguiente secuencia por pantalla (siempre y cuando `notFour` se haya escrito correctamente, claro):

```
10
8
6
-4
2
```

La relación entre un intérprete, un compilador y una máquina abstracta debe ser la siguiente: el resultado de evaluar una expresión *e* en un estado *s* debe dar como resultado un entero, el cual debe coincidir con el entero en la cima de la pila del Matarile 69000 al terminar la máquina de ejecutar las instrucciones generadas por el compilador para esa misma expresión *e* en el mismo estado *s*.

Se Pide:: Definir la función:

```
conmuta :: Estado -> Exp -> Bool
```

que dado un estado y una expresión, devuelve `True` si se cumple la relación mencionada, o `False` en caso contrario.

Ejercicios para nota (1 punto)

Dado que la práctica les parecerá corta y estarán ansiosos por añadir más cosas, les propongo las siguientes mejoras. Para obtener el punto bastaría con tener algo de código correcto en alguno de estos apartados.

Análisis contextual: añadir código relativo al análisis contextual: un comprobador de tipos para expresiones y comandos, asi como una función que compruebe que las variables utilizadas en comandos y expresiones siempre tienen un valor antes de usarse.

Extender el lenguaje: añadir la sintaxis abstracta para tener operadores unarios, valores booleanos diferentes de los enteros (tipo, literales, expresiones), y funciones locales.

Extender el compilador: añadir el código que permite compilar todo GuarriC y a ser posible GuarriC más las extensiones del apartado anterior.

Entender la potencia: Indicar si GuarriC es Turing-completo. Razonar en caso positivo, y en caso negativo añadir lo necesario al lenguaje para hacerlo Turing-completo.