Depuración de Programas Haskell a partir de Especificaciones PRE/POST en Haskell

J. M. Burgos, J. Galve, J. García, M. Sutil
Dept. Lenguajes y Sistemas Informáticos (LSIIS)
Universidad Politécnica de Madrid
28660 Boadilla del Monte - Madrid
e-mail: {jmburgos, jgalve, juliog, msutil}@fi.upm.es

Resumen[1]

En la siguiente propuesta[2] se presenta una versión inicial de la herramienta de depuración declarativa llamada *H4D2*[3], la cual forma parte de un proyecto de investigación aplicado a la innovación educativa [8].

El objetivo principal de este trabajo es ofrecer herramientas metodológicas e instrumentales que faciliten el uso de metodos formales en un primer curso de programación en Haskell [5].

1. Un depurador de programas a partir de las Especificaciones PRE/POST

1.1. Estructura del Depurador *H4D2*

La herramienta *H4D2* debe permitir al operar en los modos *depuración* / *no_depuración*. Para hacer esto posible, la implementación incluye una constante debug que actúa a modo de *flag* para depurar o no. Para pasar de un modo a otro, se debe cambiar el valor de esta constante (True/False).

```
-- Flag de depuracion
debug :: Bool
debug = True
```

```
-- Mensaje de error en PRECONDICION
error_pre :: String -> String
error pre nombreFuncion =
 "ERROR EN PRECONDICION: " ++ nombreFuncion

-- Mensaje de error en POSTCONDICION
error_post :: String -> String
error_post nombreFuncion =
"ERROR EN POSTCONDICION: " ++ nombreFuncion
```

El modo en que opera el depurador declarativo H4D2 sigue la secuencia lógica definida por la estructura de las especificaciones PRE/POST. Así, dada una función Haskell con la estructura descrita en la sección 3.1, la ejecución de la depuración seguirá la siguiente secuencia de evaluaciones:

DEPURADOR (pre, funcion, post) (x) =

Paso 1) *EVALUAR* pre (x)
Paso 2) resultado = *EVALUAR* funcion (x)
Paso 3) *EVALUAR* post (x, resultado)

Para la implementación, se ha optado por la definición de una función de orden superior depurar que actúa sobre las funciones pre, post y funcion. Se ha añadido, así mismo, un parámetro de tipo cadena de texto que representa el nombre de la función que se quiere depurar. Este último parámetro lo utilizaremos más adelante para asociar los mensajes de depuración a la función que se evalúa.

---

[1] Una versión completa de este trabajo puede encontrarse en [9].

[2] Este trabajo ha sido parcialmente financiado por el proyecto español TIC 01-2798.

[3] H*askell for* D*eclarative* D*ebugging*

```
-- Interprete de depuración
depurar :: (b -> Bool, b -> a, b -> a ->
                    Bool, String) -> b -> a
depurar (pre, funcion, post, nombre) (x) =
 eval PRE (pre (x))
   (eval_POST ((funcion x) (post x) nombre)
    nombre
```

Obsérvese, que la secuencia de evaluaciones se implementa mediante composición de funciones (eval_PRE, eval_POST). La anterior secuencia de evaluaciones puede interrumpirse si bien la precondición o la postcondición no se cumplen.

```
-- Función que evalúa la PRECONDICIÓN
eval_PRE :: Bool -> a -> String -> a
eval_PRE pre funcion nombre =
        if debug then
           if pre then funcion
           else error (error_en_pre nombre)
        else funcion

-- Función que evalua la POSTCONDICION
eval_POST :: a -> (a->Bool) -> String -> a
eval_POST funcion post nombre =
     if debug then
       if post funcion then funcion
       else error (error en post nombre)
     else funcion
```

Como puede verse, utilizamos el valor de la constante debug para evaluar en modo depuración (evaluar la precondición y la postcondición) o modo normal. Finalmente, caso de que se incumpla alguna de las condiciones, debe dispararse una excepción y finalizar la ejecución. Esto lo realizan las siguientes funciones error_pre y error_post, las cuales están parametrizadas con el nombre de la función que estamos depurando.

2. Ejemplos de Depuración

A continuación se presentan 2 ejemplos representativos del uso de la técnica propuesta para depurar programas Haskell. Los ejemplos se presentan ya con la conversión de especificación a depuración.

2.1. Ejemplo 1: *"Número Primo"*

"Determinar si un número $n \in \mathbb{Z}^+$ es primo".

Para resolver el problema definimos la función auxiliar `hay_divisor`, la cual habíamos especificado y se ha obtenido la siguiente versión:

```
import Debug

-- PROBLEMA: Determinar si en [2, x] hay
--           algún divisor de n

hay_divisor :: (Int, Int) -> Bool
hay_divisor = depurar (pre,fun,post,name)

where
--   name :: String
     name = "hay_divisor"
--   pre :: (Int,Int) -> Bool
     pre (x, n) = n > 0
--   funcion :: (Int,Int) -> Bool
     fun (1, n) = False
     fun (x, n) = (n `mod` x) ||
                  hay_divisor (x-1, n)
--   post :: (Int,Int) -> Bool -> Bool
     post (x,n) res = res ==
     (existe (ini,sig,min) (\y->True)
                           div_inf)(x,n)
--   ini :: (Int,Int) -> (Int,Int,Int)
     ini (x, n) = (2, x, n)

--   sig :: (Int,Int,Int) -> (Int,Int,Int)
     sig (inf,sup,n) = (inf+1,sup, n)

--   min :: (Int,Int,Int) -> Bool
     minimal (inf,sup,_) = (inf > sup)

--   div_inf :: (Int,Int,Int) -> Bool
     div_inf (inf,sup,n) = divide (inf,n)
```

Ahora ya, describimos la versión de depuración del problema de calcular si un número es primo o no, como sigue:

```
-- PROBLEMA: Determinar si un número n es
--           primo.

es_primo :: Int -> Bool
es_primo = depurar (pre, fun, post, name)
  where
-- name :: String
   name= "es_primo"
-- pre :: Int -> Bool
   pre n = n > 0
-- fun :: Int -> Bool
   fun n = (n == 1) || (n == 2) ||
           not (hay_divisor (n `div` 2, n))
-- post :: Int -> Bool -> Bool
   post n res =
res ==
       n==1
    || n==2
    ||(todos (ini,sig,min)(\x -> True)
                           no_divide n)
-- ini :: Int -> (Int, Int)
   ini n = (n-1, n)
-- sig :: (Int, Int) -> (Int, Int)
```

```
   sig (x, n) = (x-1, n)
-- min :: (Int, Int) -> Bool
   min (x, n) = (x == 1)
-- no divide :: (Int, Int) -> Bool
   no divide (x, n) = (n `mod` x /= 0)
```

2.2. Ejemplo 2: "Operaciones con Dígitos"

a) *"Determinar los anillos de un dígito"*

```
-- PROBLEMA : Determinar el número de
              anillos de un dígito
anillos_digito :: Int -> Int
anillos_digito = depurar(pre,fun,post,name)
where
--   name :: String
     name = "anillos_digito"
--   pre :: Int -> Bool
     pre (n) = n >= 0
--   fun :: Int -> Int
     fun (n)  = if (n == 8) then 2
                else if (n == 0 ||
                         n == 6 ||
                         n == 9) then 1
                else 0
-- post :: Int -> Int -> Bool
   post n resultado = resultado == anillos
      where anillos  | (n == 8)       = 2
                     | (n == 0) ||
                       (n == 6) ||
                       (n == 9)       = 1
                     | otherwise      = 0
```

b) *"Determinar los anillos de un número $n \in \mathbb{N}$"*

```
-- PROBLEMA : Determinar los anillos de
              un número Natural n.
anillos_numero :: Int -> Int
anillos_numero = depurar(pre,fun,post,name)
  where
--   nombre :: String
     nombre = "anillos_numero"
--   pre :: Int -> Bool
     pre (n) = n >= 0
--   fun :: Int -> Int
     fun n = if (n < 10) then
               anillos_digito (n)
             else
               anillos_digito (n `mod` 10) +
               anillos_numero (n `div` 10)
--   post :: Int -> Int -> Bool
     post n resultado =
     resultado ==
     suma (inicial, siguiente, minimal)
          (\x -> True) anillos_digito_i (n)
--   ini :: Int -> (Int, Int)
     ini (n) = (n, numDigitos(n))
--   sig :: (Int, Int) -> (Int, Int)
     sig (n, i) = (n, i-1)
--   min (Int,Int) -> Bool
     min (n, i) = i == 0
```

3. Conclusiones y Trabajos Futuros

En este trabajo hemos presentado una propuesta de depuración de programas Haskell a partir de las especificaciones PRE/POST descritas en Haskell.

En el futuro, planeamos mejorar la herramienta H4D2 con más facilidades de depuración y elementos que hagan más amigable la depuración (trazas paso a paso, evaluación de elementos del programa, etc..).

Referencias

[1] Burgos, J.M., Galve, J., García, J. and Sutil M.: Una Taxonomía de Problemas para la Enseñanza de la Programación, *Actas del VII INFOREDU'2000*, Mayo 2000, La Habana (Cuba).

[2] Burgos, J.M., Galve, J., García, J. and Sutil M.: Diseño de Soluciones Abstractas mediante el Refinamiento de Especificaciones. VII Jornadas sobre la Enseñanza Universitaria de la Informática (JENUI'01).

[3] J.M. Burgos, J.Galve y J. García. From Problems to Programs: A Pattern Language to Go from Problem Requirements to Solution Schemas in Elementary Programming. Proceedings of the 5th EuroPLoP´2000.

[4] Schmidt, D.R. Towards a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology*, AMAST'96 (1996), vol. LNCS 1101, Springer-Verlag, pp. 62-84.

[5] http://www.haskell.org/

[6] S. Peyton Jones and J. Hughes (editors). Standard libraries for the Haskell 98 programming language, February 1999.

[7] E.Y.Shapiro. Algorithmic Program Debugging. MIT Press, Cambridge, MA, 1983.

[8] Grupo de Investigación de Programación Declarativa Aplicada LSIIS. *Herramientas metodológicas e instrumentales para la enseñanza de la Programación*. Proyecto Fundación General de la UPM (FG-UPM-43700000190), 2000-2001. Grupo de Investigación de Programación Declarativa Aplicada LSIIS. *Herramientas*

metodológicas e instrumentales para la enseñanza de la Programación. Proyecto Fundación General de la UPM (FG-UPM-43700000190), 2000-2001.

[9] http://abraham.ls.fi.upm.es/h4d2/jenui02.pdf