Haskell en toda su pureza

Manuel Hernández
manuelhg@mixteco.utm.mx
Universidad Tecnológica de la Mixteca
Huajuapan de León, Oaxaca, C.P. 6900

RESUMEN

Haskell es un lenguaje de programación funcional pura. La idea de programar funcionalmente es sólo utilizar funciones calculables como bloques constructores de programas. La idea de pureza es que todo aquello que sea posible con un lenguaje de programación debe tener una contraparte matemática básica. Paradójicamente, dentro de estas dos ideas está ésta otra: facilitar buenas técnicas de programación sin que el programador acuda a las matemáticas en sí.

1. INTRODUCCIÓN

Haskell es un lenguaje de programación que tiene como bloques constructores *funciones*. Las funciones son fundamentales en las Matemáticas, pero su utilización como artefactos computacionales es un poco sorprendente. De manera curiosa, las funciones fueron ya utilizadas como un lenguaje de programación en los primeros tiempos de las indagaciones del poder computacional vía Máquinas de Turing, allá por los años 30s del siglo pasado. Este tutorial está destinado a que conozcas un poco del arte y la ciencia de la Programación Funcional.

2. FUNCIONES

Comenzemos ahora con el concepto *matemático* de función. Sea `f` una función matemática. Esta función tiene un *dominio* y un *codominio*. Esto es conocido como el *tipo* de una función.

Ahora pensemos en funciones que se pueden calcular: aquellas para las que existe un *algoritmo* que nos permite que de la función, aplicada a un argumento, se obtenga un resultado. Esta noción de entrada-salida es tan importante que no debes dudar en poner manos a la obra, bajar el intérpre de Hugs de la Interner (¡es gratis! y hay versiones para Mac, Linux, y Windows), y con tus notitas y tu intérprete *ejecutar* las funciones en discusión.

Consideremos la siguiente función recursiva de las matemáticas, la función de Fibonacci:

Material de Haskell: [2.1]

```
1 fib :: Integer -> Integer
2 fib 0          = 1
3 fib 1          = 1
4 fib n | (n>1) = fib (n - 1) + fib (n - 2)
5 fib _          = error "fib no definida"
```

Analicemos inmediatamente esta función en sus componentes básicos. La definición de una función en Haskell consta:

1. De un nombre asociado a esta función (`fib`, en este caso, pues pueden existir funciones sin nombre);
2. de un número de posibles argumentos que puede tener la función (la *aridad* de la función);
3. de un posible tipo asociado a la función (línea 1) (la *signatura* de la función);
4. del dominio y el codominio de la función explícitamente establecido (en este caso, del tipo `Integer` para dominio y codominio);
5. de un posible conjunto de *ecuaciones dirigidas* que tratarán el caso apropiado para cuando esta función sea invocada;
6. en cada ecuación habrá un lado derecho (el *cuerpo* de la función) que permitirá establecer qué es lo que hace la función; en este cuerpo son posibles las llamadas a las funciones preconstruidas, las definidas por el usuario, y quizás llamadas a la misma función que se está definiendo (recursividad);
7. de un posible manejo de excepciones, que aunque teóricamente no tienen que establecerse, en la práctica pueden ser fundamentales; en este caso, en la línea 5 manejamos un error que ocurriría cuando la llamada a la función `fib` no es la que se podría esperar (con un número entero positivo, en el caso ideal).

Con el símbolo _ estamos tratados los casos que se nos hayan pasado por alto en nuestra definición. ¿Cómo es que Haskell "sabe" qué argumento es el apropiado para tomarlo como argumento? Por un mecanismo de *casamiento de patrones* (*pattern matching*, en inglés), que discrimina algorítmicante la estructura o el valor de la expresión con la que la función fue invocada. Posteriormente hablaremos un poco más de este tema. Podemos reformular nuestra definición para hacerla menos dependiente del casamiento de patrones, omitiendo de ahora en adelante el manejo de excepciones:

Material de Haskell: [2.2]

```
fib :: Integer -> Integer
fib n = if n==0 || n==1 then 1
                else
                     fib (n - 1) + fib (n - 2)
```

o inclusive

Material de Haskell: [2.3]

```
fib :: Integer -> Integer
fib n = case n of
                0   -> 1
                1   -> 1
          otherwise -> fib (n - 1) + fib (n - 2)
--Este es un comentario en Haskell, por cierto
```

o como

Material de Haskell: [2.4]

```
fib :: Integer -> Integer
fib n | n==0 || n==1 = 1
      | otherwise = fib (n - 1) + fib (n - 2)
```

O todavía peor:

Material de Haskell: [2.5]

```
fib :: Int -> Integer
fib n = fibs !! n
  where
    fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

o bien

Material de Haskell: [2.6]

```
fibs=[1,1]++[n+m|
           (n,m)<-(zip fibs (tail fibs))]
fib n = fibs !! n
```

Variando un poco el tema, en la siguiente sesión podemos de hecho terminar con las *cadenas de Fibonacci*:

Material de Haskell: [2.7]

```
fibs  = [1,1]++[m|
           m<-(zipWith (+) fibs (tail fibs))]
test1 = take 6 fibs
--[1,1,2,3,5,8]

fibs1 = [[1],[1]]++[ m|
           m<-(zipWith (++) fibs1 (tail fibs1))]
test2 = take 6 fibs1
--    [[1],[1],[1,1],[1,1,1],[1,1,1,1,1],[1,1,1,1,1,1,1,1]]

fibs2 = [[1],[2]]++[ m|
           m<-(zipWith (++) fibs2 (tail fibs2))]
test3 = take 6 fibs2
--    [[1],[2],[1,2],[2,1,2],[1,2,2,1,2],[2,1,2,1,2,2,1,2]]

fibs3 = ["a","b"]++[m|
           m<-(zipWith (++) fibs3 (tail fibs3))]
test4 = take 6 fibs3
--["a","b","ab","bab","abbab","bababbab"]
```

La utilización del signo `=` es potencialmente confusa, ya que se puede interpretar como la relación de igualdad en Matemáticas. En este caso, esta interpretación es aproximada, ya que nos permite definir funciones leyendo como que la parte derecha del símbolo de igualdad se identifica con la parte izquierda, pero que sintácticamente lo contrario no es posible (no hay ley simétrica para el símbolo `=`). Esta es la forma usual cuando decimos: "Sea f(x)=x*x". En la siguiente Sección analizamos qué tipo de argumentos son válidos para una función.

3. TIPOS DE DATOS SIMPLES

Consideremos primero un pequeño mundo de colores:

Material de Haskell: [3.1]

```
data = Blanco | Negro | Rojo | Amarillo | Azul
     | Verde | Violeta | Naranja
```

La palabra reservada `data` nos permite definir *datos y estructuras de datos*. En este caso, el dato `Color` sólo tiene 8 posibles casos. Consideremos una función llamada `mezcla`, que toma un argumento en forma de par ordenado y tiene como dato de salida el tipo `Color`

Material de Haskell: [3.2]

```
mezcla :: (Color,Color) -> Color
mezcla (Rojo, Amarillo) = Naranja
mezcla (Rojo, Azul)     = Violeta
mezcla (Amarillo, Azul) = Verde
```

Notemos que `mezcla` es una función parcial sobre el tipo `Color`, pues por ejemplo la evaluación de
`mezcla (Negro, Blanco)`
no está definida. Está sería la primera modificación del programa anterior:

Material de Haskell: [3.3]

```
mezcla :: (Color,Color) -> Color
mezcla (Rojo, Amarillo) = Naranja
mezcla (Rojo, Azul)     = Violeta
mezcla (Amarillo, Azul) = Verde
mezcla (_,_)            = error "mezcla no definida"
```

Como ya comentamos, el símbolo de subrayado `_` actúa como una variable "atrapa-todo" (del resto de los valores definidos en `data`), en tanto que la palabra reservada `error` llama a una *excepción* de manejo de errores.

Otra modificación práctica está condicionada por la utilización de Haskell: Haskell no es capaz de imprimir en pantalla los componentes del dato `Color`. Esto se remedia haciendo que `Color` se derive de una clase especial en Haskell llamada `Show`:

Material de Haskell: [3.4]

```
data Color = Blanco | Negro | Rojo | Amarillo
           | Azul | Verde | Violeta | Naranja
                          deriving Show

mezcla :: (Color, Color) -> Color
mezcla (Rojo, Amarillo) = Naranja
mezcla (Rojo, Azul)     = Violeta
mezcla (Amarillo, Azul) = Verde
mezcla (_,_)            = error "mezcla no definida"
```

Haskell propone la estructuración de datos por medio de *clases*, que consisten en datos y funciones, con mecanismos parecidos a los de la Programación Orientada a Objetos.

Otros tipos de datos importantes son los siguientes: `Integer`, `Int`, `Float`, `String`, `Char`, y `Bool`.

4. TIPOS DE DATOS RECURSIVOS

En el siguiente programa analizamos la palabra reservada `data` y su utilización en la definición de *datos recursivos*. A diferencia de otros muchos lenguajes de programación en donde en general se desanima la recursión, en Haskell tanto los datos como las funciones se pueden definir sin problemas en versiones naturalmente recursivas.

Introducimos la idea detrás de un tipo de dato recursivo:

Material de Haskell: [4.1]

```
data Nat = N Nat | Zero deriving Show

suma_nat Zero a = a
suma_nat (N a) b = N (suma_nat a b)
```

Hemos ya mencionado que la palabra reservada `data` es una palabra especial que nos permite *definir* algunos tipos de datos. En este caso, tenemos dos posibles tipos de datos: o bien un tipo `N n`, donde `n` pertenece al tipo de datos `Nat`, números naturales, o `Zero`. Esta distinción entre la estructura básica de un tipo de dato se escribe con el símbolo `|`. Notemos que `Nat` se llama a sí mismo en su definición. Algunos ejemplos de elementos en `Nat` son los siguientes: `Zero, N (Zero), N (N (Zero)), N (N (N (Zero)))`. Una modelación semántica natural de estos datos sería identificarlos con los números naturales y el cero.

En ...`deriving Show` nuestra intención es *mostrar* el dato `Nat`: `Show` es una *clase* especial de Haskell diseñada para mostrar objetos que son una composición de datos primitivos.

Los tipos de datos *recursivos* se definen por medio de datos simples (en situaciones terminales) y a sí mismos (en los casos recursivos). Un caso típico es el de una lista: una lista puede ser una lista vacía (una lista sin elementos) o tener una cabeza al frente de una lista. Antes de entrar en los tipos de datos *polimórficos* comenzaremos hablando únicamente de listas de enteros. La definición en Haskell es así:

Material de Haskell: [4.2]

```
data LInt = Nil | Cons (Int, LInt)
```

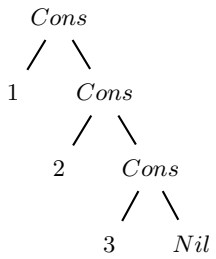

Figura 1: Una lista.

Estos tipos de datos son generalmente la entrada a funciones que operan sobre cada componente. Por ello, las funciones asociadas son también recursivas.

En efecto, notamos que el tipo de datos `LInt` (lista de enteros) es recursivo: Una lista de enteros puede ser la lista vacía (`Nil`), y si no lo es, entonces es una posible anidación del identificador `Cons` cierta cantidad de veces (por ejemplo, `Cons(1, Cons(2, Cons(3,Nil)))`). Esto puede representarse por el siquiente árbol binario en forma de peine, dado en la Fig. 1. Como una notación más conveniente, utilizaremos la notación de lista: `Cons(1, Cons(2, Cons(3,Nil)))` es representado como `[1,2,3]`, mientras que `Nil` es representado por `[]`. Además, dado un tipo de dato `D`, el hecho de que una función `f` toma como argumento una lista de elementos del tipo `D` y retorna un valor `E` se escribe así:

```
f :: [D] -> E
```

Este es un ejemplo de una función *polimórfica*. Una función polimórfica tiene como dominio la unión de varios tipos de datos, y se adapta al dato con el cual es llamada. Veremos ejemplos de este concepto más adelante.

Consideremos por ejemplo la siguiente definición que nos permite hallar la longitud de una lista:

Material de Haskell: [4.3]

```
lengthlist  :: [a] -> Int
lengthlist  []     = 0
lengthlist  (a:bs) = 1 + lengthlist  bs
```

Con la expresión `[a]` señalamos que la naturaleza de los elementos de la lista es irrevelante para calcular la longitud de la lista (sin embargo, todos los elementos deben ser del mismo tipo; las listas con esta característica se llaman *homogéneas*). También es importante aclarar que la variables `[a]` sobre tipos de la primera línea no tiene nada que ver con la variable a de la tercera línea. Sucede que en la comunidad funcional `[a]` representa $[\alpha]$ (o las primeras letras del alfabeto griego) para denotar la variabilidad en tipos.

La expresión `(a:bs)` es un *deconstructor*: cuando `(a:bs)`, por un mecanismo llamado *análisis estructural* (*pattern matching*, en inglés), se iguala a la lista `[1,2,3]`. Convengamos en denotar por el operador $\triangle$ una operación que se puede leer en $l \triangle r$ "r intenta analizarse por medio de l". Como aquí estamos tratando el tema matemáticamente (y no en

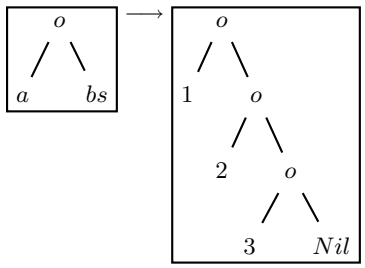

Figura 2: Una abstracción nombrada en un árbol.

Haskell), analizamos la situación en la siguiente expresión matemática:

$$(a : bs)\triangle[1, 2, 3]$$

(el caso $[]\triangle[1, 2, 3]$ queda descartado, pues la lista $[1, 2, 3]$ es no vacía). Para la expresión dada, a adopta el valor 1, y bs el valor $[2, 3]$. El valor que ha tomado bs es la *cola* de la lista $[1, 2, 3]$. En la Fig. 2 representamos gráficamente la identificación abstracta entre los árboles involucrados.

El mecanismo de análisis estructural aquí bosquejado es en realidad un algoritmo que nos permite decidir si una expresión casa con algún tipo predefinido, o definido por el usuario, de "molde". El mecanismo funciona como sigue:

1. si x es una variable, y a una constante, $x\triangle a$ permite que x se identifique con a si a es una constante y x una variable (además, se dice que x queda *ligada a* a);
2. si a y b con constantes, $a\triangle b$ hace que el algoritmo falle si $b \neq a$, y tenga éxito inmediatamente si $b = a$;
3. $c(d_1, d_2)\triangle e(g_1, g_2)$ induce a investigar los casos $d_1\triangle g_1$ y $d_2\triangle g_2$ si $c = e$; si la investigación de los casos presentes tiene éxito, el algoritmo tiene exito, de lo contrario falla. Falla también inmediatamente si $c \neq e$.

Aquí hemos hablado de una constante como un valor normalizado.

5. CONSTRUCCIONES BÁSICAS EN HASKELL

A continuación describimos algunas construcciones básicas de Haskell.

5.1 Los guardias

Como primera construcción básica de Haskell, exploremos el uso de *guardias* para *restringir* los posibles argumentos de una función. Ejemplifiquemos esta noción de restricción con la siguiente definición del valor absoluto de un número:

Material de Haskell: [5.1]

```
abs n | n==0 = 0
abs n | n<0 = −n
abs n | n>0 = n
```

La expresión `abs n | n>0 = n` se lee como "el valor absoluto de n es n si n es mayor que 0". La parte condicional radica en `|`: los argumentos quedan *filtrados* por medio de la condición a la derecha de `|`. La introducción de paréntesis puede explicar más la idea:

Material de Haskell: [5.2]

```
abs n | (n==0) = 0
abs n | (n<0) = −n
abs n | (n>0) = n
```

5.2 La condicional

Una *expresión condicional* se ejemplifica del siguiente modo, con el mismo ejemplo del valor absoluto:

Material de Haskell: [5.3]

```
abs n = if (n>=0) then n
                  else −n
```

Como vemos, `if`, `then`, y `else` son palabras reservadas de Haskell. Despues de una condición booleana que tiene que ir entre `if` y `then`, la primera rama de la condicional está entre `then` y `else`, y la segunda rama viene despues de `else`. A diferencia de Scheme, aquí la construcción `if` tiene que constar de un `then` y de un `else`.

5.3 La construcción de análisis de casos

La construcción de `case` es un poco más complicada que la de `if`:

Material de Haskell: [5.4]

```
traduce n = case n of
                1  −> "uno"
                2  −> "dos"
                3  −> "tres"
                _  −> "desconocido"
```

La palabra `case` tiene que ir junto con la palabra `of`, y de preferencia los casos deben ser de tipo simple (si los casos son booleanos, `true` y `false`, y si numéricos, números simples, digamos); de lo contrario, es mejor utilizar la construcción `if`. Las flechas corresponden a la evaluación del caso que se espere. El símbolo de subrayado `_` trata casos no previstos, *del mismpo tipo que los anteriores.*

5.4 Las construcciones de abstracción

Son dos las construcciones de abstracción en Haskell: la construcción `where` y la construcción `let`. Las discutimos a continuación.

5.4.1 Nombramientos retardados

La construcción `where` es una importante construcción de abstracción que permite primero utilizar *nombres de variables* y *posteriormente* conocer a qué se ha nombrado con estas variables.

Consideremos el siguiente problema: deseamos hallar las raíces reales de una *ecuación cuadrática*:

$$a * x^2 + b * x + c = 0$$

Como sabemos, las raíces reales existen si el *discriminante* es mayor que o igual a 0. Si tal es el caso, las raíces son:

$$\frac{-b^2 - \sqrt{b^2 - 4 * a * c}}{2 * a} \quad (1)$$

$$\frac{-b^2 + \sqrt{b^2 - 4 * a * c}}{2 * a} \quad (2)$$

Dejando parte de trabajo al mecanismo de reportes de errores de nuestro intérprete, la solución directa al problema planteado sería la siguiente:

Material de Haskell: [5.5]

```
1 roots :: Float -> Float ->Float -> (Float, Float)
2 roots a b c = ((-b+sqrt(b^2-4*a*c))/(2*a),
3                (-b-sqrt(b^2-4*a*c))/(2*a))
```

(Notemos la forma multifuncional de `roots`).

Como primera abstracción, el discriminante podría ser nombrado de forma retardada:

Material de Haskell: [5.6]

```
1 roots :: Float -> Float ->Float -> (Float, Float)
2 roots a b c = ((-b+sqrt d)/(2*a), (-b-sqrt d)/(2*a))
3                 where d = b^2-4*a*c
```

Otra abstracción podría realizarse sobre la expresión `2*a`, y obtendríamos:

Material de Haskell: [5.7]

```
1 roots :: Float -> Float ->Float -> (Float, Float)
2 roots a b c = ((-b+sqrt d)/v, (-b-sqrt d)/v)
3                 where d = b^2-4*a*c
4                       v = 2*a
```

Notemos el importante papel que juega la disposición (*layout*) de las expresiones en Haskell. Inclusive es posible realizar un pequeño manejo de error para anunciar (sin calcular) las raíces complejas:

Material de Haskell: [5.8]

```
1 roots :: Float -> Float -> Float -> (Float, Float)
2 roots a b c = if (d>=0)
3                  then
4                    ((-b+sqrt d)/v, (-b-sqrt d)/v)
5                  else error "Raíces complejas"
6                       where d = b^2-4*a*c
7                             v = 2*a
```

Destacamos que las variables definidas en `where` son locales a la definición de la función (de hecho, a cada ecuación que define la función).

5.4.2 Nombramientos adelantados

Ahora veremos un mecanismo de nombramiento adelantado: la construcción `let`.

Material de Haskell: [5.9]

```
1 ejemplolet = let x="Primero-"
2                  y="Segundo-"
3                  z="Tercero" in
4                      x++y++z
```

Esta construcción nos permite nombrar elementos antes de utilizarlos. Los nombres de las variables son locales a la función.

Como ejemplo complementario podríamos utilizar también la construcción `let` en el problema de encontrar las raíces reales de la ecuación cuadrática:

Material de Haskell: [5.10]

```
1 roots2 :: Float -> Float -> Float -> (Float, Float)
2 roots2 a b c = let
3                  d = b^2-4*a*c
4                  v = 2*a
5                    in
6                       if (d>=0)
7                          then
8                          ((-b+ sqrt d)/v, (-b-sqrt d)/v)
9                          else error "Raíces complejas"
```

5.5 Listas intensionales o compactas

En Haskell es posible utilizar listas que no necesariamente son "finitas". Además, existen notaciones especiales para describir sucintamente algunas listas que tienen una estructura regular. Las listas así descritas se conocen como listas *intensionales o compactas*. Por ejemplo, la listas compactas en

```
1 [1..]
2 [1,2..]
3 [1,1.1..]
4 [1,1.01..2]
```

denotan las siguientes listas: `[1..]` denota una lista infinita con elementos que se incrementan en cada ocasión una unidad, por lo que `[1..]`, de desglosarse en el listado de sus primeros elementos, sería la lista `[1,2,3,4,5,6,7,...]`. La lista del renglón 2 es esta misma lista. La lista del renglón 3 es una lista que se incrementa 0.1 ($1.1 - 1 = 0.1$) en cada ocasión. El valor 0.1 es el *incremento* de la lista. Esta lista es también *infinita*. Finalmente, la lista del renglón 4 tiene un incremento de 0.01, pero es finita, y termina cuando alcanza el valor 2, y en otros casos parecidos, antes de que sobrepase este valor. También es posible que si una lista intensional no tiene coherencia en su definición produzca simplementen la lista vacía.

Podríamos dar una descripción abstracta de cada caso anterior, pero sólo presentamos el caso de la línea 4: En $l = [a, b..c]$, con $a \leq b$, l es una lista *finita* con a como el *límite izquierdo*, $b - a$ es el *incremento*, y b es el *límite derecho*.

En el siguiente segmento de sesión ilustramos algunas ideas relacionadas con las listas intensionales:

```
1 Hugs session for:
```

```
/usr/local/lib/hugs/lib/Prelude.hs
Type :? for help
Prelude> take 5 [1..]
 [1,2,3,4,5]
Prelude> [1,1.1..2]
    [1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2.0]
Prelude> [1,5..2]
 [1]
Prelude> [1,-5..2]
 []
Prelude> [-5..2]
[-5,-4,-3,-2,-1,0,1,2]
Prelude> take 5 [1,1.001..2]
 [1.0,1.001,1.002,1.003,1.004]
Prelude> map sin (take 5 [1,1.001..2])
 [0.8414,0.8420,0.8425,0.8430,0.8436]
Prelude> zip [1,1.1..2] (map sin (take 4 [1,1.1..2]))
   [(1.0,0.8414),(1.1,0.8912),(1.2,0.9320),(1.3,0.9635)]
```

(Una palabra diferente a `Prelude` aparecerá en la línea de comandos —*prompt*— cuando cargues correctamente un archivo.)

5.6 Abstracciones lambda y funciones puras

En Haskell también podemos tratar con funciones llamadas *funciones lambda* o *funciones anónimas*.

```
map (\x -> sin x) [1,2,3]
map (\x -> x^2) [1..10]
```

Similarmente, la siguiente definición es una forma *pura* de la función `sqr`.

Material de Haskell: [5.11]

```
sqr = (\x -> x*x)
```

Notemos que esta función ni siquiera tiene argumentos en el lado izquiera Otras funciones que se definirían sin argumentos serían:

Material de Haskell: [5.12]

```
sc=sin . cos
sum = foldr (+) 0
```

Dos usos posibles serían:

```
Prelude> let sc=sin . cos in sc 1
0.514395
Prelude> let sum = foldr (+) 0 in sum [1,1.1..5]
123.0
```

Este estilo de definir y utilizar funciones se le conoce como *libre de puntos*, del inglés *point-free*.

6. FUNCIONES COMUNES

Algunas funciones comunes en la Programación Funcional son las que a continuación describimos (ver el archivo `Prelude.hs` en la distribución de Hugs). Estas funciones es posible utilizarlas libremente en la construcción de programas en Haskell, pero es necesario asegurarse que el archivo `Prelude.hs` ha sido cargado (lo que generalmente acontece en las implementaciones de intérpretes de Haskell). Si el archivo ha sido cargado, redefinir estas funciones nos causará problemas. Si es necesario experimentar con estas funciones lo mejor es usar otro nombre diferente al utilizado en el código de `Prelude.hs`.

Damos otra definición relativa a funciones, ya que la necesitaremos en algunos de estos ejemplos. Sea `(op) :: a->b->c` una función de dos argumentos. Con `(op)` indicamos que la función puede escribirse en notación infija: `m op n`. Una *sección* de una función es la función que resulta de omitir uno de sus argumentos, y fijar el otro. Por ejemplo, si `u` tiene tipo `b`, y `v` tiene tipo `a`, la anterior función tiene dos secciones: `(op u) ::a->c`, y `(v op) :: b->c`.

1. Esta función es la *primera proyección* de un par ordenado:

   Material de Haskell: [6.1]

   ```
   fst :: (a,b) -> a
   fst (x,_) = x
   ```

2. La siguiente función es la *segunda proyección* de un par ordenado:

   Material de Haskell: [6.2]

   ```
   snd :: (a,b) -> b
   snd (_,y) = y
   ```

3. a continuación se define la función `.` para la composición de funciones:

   Material de Haskell: [6.3]

   ```
   (.) :: (b -> c) -> (a -> b) -> (a -> c)
   (f . g) x = f (g x)
   ```

   Esta función se utiliza implícitamente en variadas ocasiones en Programación Funcional, y su importancia radica en una división de tareas que frecuentemente es necesario realizar en el diseño de programas grandes.

4. La función `until` nos recuerda su equivalente imperativo:

   Material de Haskell: [6.4]

   ```
   until :: (a -> Bool) -> (a -> a) -> a -> a
   until p f x = if p x then x else until p f (f x)
   ```

La siguiente función extrae la cabeza de una lista no vacía:

Material de Haskell: [6.5]

```
head :: [a] -> a
head (x:_) = x
```

Si intentamos extraer la cabeza de una lista vacía el intérprete nos enviará un mensaje de error; es importante que el

programador, en todo caso, esté prevenido para evitar problemas como esté (excepcionales, pero no impredecibles).

Estas otras funciones obtienen:

1. El último elemento de una lista no vacía:

   Material de Haskell: [6.6]

```
last :: [a] -> a
last [x] = x
last (_:xs) = last xs
```

2. la cola de una lista no vacía:

   Material de Haskell: [6.7]

```
tail :: [a] -> [a]
tail (_:xs) = xs
```

3. el *mayor segmento inicial* de una lista:

   Material de Haskell: [6.8]

```
init :: [a] -> [a]
init [x] = []
init (x:xs) = x : init xs
```

4. y la comprobación de que si una lista es o no vacía:

   Material de Haskell: [6.9]

```
null :: [a] -> Bool
null [] = True
null (_:_) = False
```

Algunas de las funciones más importante que operan sobre listas son las siguientes:

1. Dos listas homogéneas que tengan elementos del mismo tiempo pueden concatenarse para originar una tercera lista. El programa que realiza esto es el siguiente:

   Material de Haskell: [6.10]

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

2. Una función de alto orden que ya hemos conocido de nuestra experiencia con Scheme es la siguiente:

   Material de Haskell: [6.11]

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```

   Las funciones de alto orden son una parte fundamental del poder conceptual en la construcción de programas en Haskell. Tales funciones toman como entrada a otras funciones. En Haskell, la función `map` toma como argumento otra función (digamos `f`, cuyo dominio es el tipo de datos `a` y codominio el tipo de datos `b`) y una lista de elementos del tipo `a`, para devolver una lista de tipo `b`. La *signatura* de la función en el primer renglón expresa concisamente el dominio y codominio de la función `map`.

3. La siguiente función es también de mayor orden, pero esta vez el tipo de funciones que acepta está restringido a ser `a->Bool`, es decir, predicados.

   Material de Haskell: [6.12]

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]
```

   Como ejemplo,

```
>filter even [1..10]
 [2,4,6,8,10]
```

   en tanto que

```
>filter (<4) [1..10]
 [1,2,3]
```

   Debemos advertir acerca de la utilización de la función *función seccionada* `(<4)` en este ejemplo.

4. La siguiente función, una de las más útiles en programación funcional de mayor orden, toma normalmente tres argumentos: un operador asociativo a la derecha `op :: a -> b ->b`, un valor `u` de tipo `b`, y una lista `[a]`, para dar como resultado un valor `v` de tipo `b`. La selección de cada argumento está regida por algunas leyes generales. Por ejemplo, `u` normalmente debe ser un valor neutro con respecto a la operación `op`.

   Material de Haskell: [6.13]

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

   En una utilización convencional de `foldr`, la siguiente identidad matemática es válida:

   $$foldr \circ x_0\ [x_1, x_2, \ldots, x_n] = (x_1 \circ (x_2 \circ \ldots \circ x_n \circ x_0)) \ldots)$$

   Todavía de forma más particular:

   $$foldr\ \circ\ x_0\ [x_1, x_2, x_3] = (x_1 \circ\ (x_2\ \circ\ (x_3 \circ\ x_0)))$$

   Como ejemplos: `sum = foldr (+) 0`, de forma que

```
sum [1,2,3] = foldr (+) 0 [1,2,3]
            = (1+(2+(3+0))) = 6;
```

   `product = foldr (*) 1`.

   A continuación vemos otro ejemplo al definir la siguiente función.

5. Es posible tener el tipo construido siguiente `[[a]]`, que denota el tipo de datos consistente de listas de listas de elementos del tipo `a`. Una función que concatena todas las listas de una lista es la siguiente:

   Material de Haskell: [6.14]

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

6. Otra vez utilizando `foldr` algunas funciones booleanas se pueden definir así:

   Material de Haskell: [6.15]

```
and, or :: [Bool] -> Bool
and = foldr (&&) True
or = foldr (||) False
```

7. Para complementar, las funciones existenciales ∃ y ∀ (algo restringidas) tienen unas concisas definiciones como sigue:

   Material de Haskell: [6.16]

```
any, all  :: (a -> Bool) -> [a] -> Bool
any p = or . map p
all p = and . map p
```

8. Sería difícil negar la calidad declarativa de la siguiente definición de pertenencia y no pertenencia a una lista:

   Material de Haskell: [6.17]

```
elem, notElem :: Eq a => a -> [a] -> Bool
elem = any . (==)
notElem = all . (/=)
```

9. La siguiente función aplica una función `f` a un valor `x` tanto como deseemos.

   Material de Haskell: [6.18]

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

   Esta función normalmente se utiliza bajo evaluación perezosa. Por ejemplo:

```
>take 3 (iterate sqr 2)
 [2,4,16]
```

   en donde `sqr x = x*x`.

   La función siguiente, `repeat`, es también frecuentemente utilizada en evaluación perezosa (al evaluar aisladamente `repeat 2` un intérprete presentará números 2 hasta que el usuario oprima control-c).

   Material de Haskell: [6.19]

```
repeat :: a -> [a]
repeat x = xs where xs = x:xs
```

10. La siguiente es la definición "oficial" de `take`:

    Material de Haskell: [6.20]

```
take :: Int -> [a] -> [a]
take n _  | n <= 0 = []
take _ []  = []
take n (x:xs) = x : take (n-1) xs
```

11. Las definiciones también "oficiales" de `sum` y `product` utilizan una versión un poco diferente de iteración a la antes dada (que fue con `foldr`): `foldl` esta vez trabaja con operadores binarios asociativos a la izquierda. Por lo demás, hemos de notar la estructura de la definición del dominio y del codominio.

    Material de Haskell: [6.21]

```
sum, product :: Num a => [a] -> a
sum = foldl (+) 0
product = foldl (*) 1
```

12. La función `zip` toma dos listas, y devuelve una lista de pares ordenados. Su nombre viene de su analogía a un cierre (de *zipper*, en inglés).

    Material de Haskell: [6.22]

```
zip :: [a] -> [b] -> [(a,b)]
zip = zipWith (\a b -> (a,b))
```

    La función *inversa* es `unzip`.

13. Las siguientes funciones también son muy utilizadas en Programación Funcional:

    Material de Haskell: [6.23]

```
drop :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs
drop _ []  = []
drop n (_:xs) = drop (n-1) xs

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
          | p x = x : takeWhile p xs
          | otherwise = []
```

    El siguiente código es un poco dual al anterior:

    Material de Haskell: [6.24]

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p xs@(x:xs')
          | p x = dropWhile p xs'
          | otherwise = xs
```

    En el intérpret Hugs, ambas funciones se ejemplifican como:

    Material de Haskell: [6.25]

```
Hugs> takeWhile (<10) [1..]
  [1,2,3,4,5,6,7,8,9]
Hugs> dropWhile (<10) [1..20]
  [10,11,12,13,14,15,16,17,18,19,20]
```

7. TIPOS DE DATOS INFINITOS

Definimos ahora una función que es común en Programación Funcional, la función `take`:

Material de Haskell: [7.1]

```
take :: Int -> [a] -> [a]
take 0 ls  = []
take n (a:ls ) = (a : (take n ls))
```

El siguiente programa parece absurdo:

Material de Haskell: [7.2]

```
ones=1:ones
```

Sin embargo, consideremos la expresión `take 3 ones`: si la estrategia de evaluación fuera la normal, primero intentaríamos evaluar a 3 y a `ones`. La constante 3 está en forma normal y se evalúa a 3. La expresión `ones`, sin embargo, no puede evaluarse (entra en un ciclo infinito), y por lo tanto la expresión completa `take 3 ones` no puede evaluarse. Consideremos ahora la misma expresión `take 3 ones` evaluada con la estrategia de evaluación perezosa: en esta ocasión, primero tenemos que evaluar la función `take`. La función `take` es estricta en su primer argumento, pero su primer argumento está ya evaluado. Sin embargo, el deconstructor `(a:bs)` en la tercer línea sí requiere conocer los valores de que consta `ones`. En la siguiente expresión denotamos por → un paso de evaluación que sigue la estrategia de evaluación perezosa:

`take 3 ones` → `take 3 (1:ones)` → `(1:(take 2 ones))`

Requerimos ahora de la evaluación de `take 2 ones`:

`take 2 ones` → `take 1 (1:ones)` → `(1:(take 0 ones))`

Finalmente, requerimos de la evaluación de `take 0 ones`:

`take 0 ones` → `[]`

Ahora en `take 0 ones` ya no es posible simplificar más, pues está expresión se evalúa a `[]` independientemente del valor de `ones` (ver línea 1).

Por todo lo anterior, la evaluación de la expresión `take 3 ones` es `[1,1,1]`, a pesar de que uno de sus argumentos no es finito.

8. LISTAS COMPACTAS

Una *lista compacta* es una lista que nos recuerda bastante la construcción intensional de conjuntos. En la siguiente expresión de Haskell

```
[x | x<-ls, Q1(x), Q2(x), Q3(x)]
```

`ls` representa una lista (finita o infinita), y `Q1`, `Q2`, y `Q3` son predicados. La expresión `x<-ls` se llama un *generador*, en tanto que las expresiones `Q1`, `Q2`, y `Q3` son *cualificadores* o *restrictores*.

Una lista compacta es una forma sencilla de brindar múltiples evaluaciones a la vez. Por ejemplo, consideremos la siguiente definición de *triadas pitagóricas*:

Material de Haskell: [8.1]

```
1 triadas n = [(x,y,z) | x<−[0..n],
2                        y<−[0..n],
3                        z<−[0..n],
4                        x*x+y*y==z*z]
```

Lo conciso y claro de esta definición ejemplifica la importancia notacional de las listas compactas.

En el siguiente programa ejemplificamos otra vez la utilización de las listas compactas, al resolver el *problema de las 8 reinas*, que consiste en colocar 8 reinas sin que ninguna de ellas ataque a la otra en un tablero de ajedrez.

Material de Haskell: [8.2]

```
1 queens nqueens = qu nqueens where
2     qu 0     = [[]]
3     qu (m+1)= [p++[n]|p<−qu m,n<−[1..nqueens],
4                                  safe p n ]

6 safe p n  = all not [check (i,j) (m,n)|(i,j)<−zip [1..] p]
7                     where m = 1+length p

9 check (i,j) (m,n) = j==n || (i+j==m+n) || (i−j==m−n)
```

Con la solicitud de la evaluación `queens 8` resolveremos el problema de las 8 reinas, expresando todas sus posibles soluciones (con redundancias en simetría).

Notamos que la lista compacta definida en la línea 3 consta de 2 generadores y un restrictor.

9. ORDENAMIENTO: ALGUNOS ALGORITMOS

El siguiente es un problema ubicuo y clásico de la programación de computadoras: Escriba un programa que ordene una sucesión finita de números (que no necesariamente son todos diferentes). Diversos algoritmos se han ideado para dar solución a este problema. A continuación escribiremos en programación funcional los algoritmos más relevantes (y hay muchos de ellos) que resuelven el problema de ordenamiento.

9.1 El algoritmo de ordenamiento rápido

El algoritmo de ordenamiento rápido tiene una definición sorprendentemente simple en Haskell:

Material de Haskell: [9.1]

```
1 qsort :: (Ord a) => [a] −> [a]
2 qsort []=[]
3 qsort (a:bs) = qsort [x | x<−bs,x<=a]
4                      ++ [a]
5                      ++ qsort [x | x<−bs,x>a]
```

La expresión `qsort :: (Ord a) => [a] -> [a]` se lee como sigue: “qsort ordena listas con elementos del tipo a si estos elementos son comparables en algún orden”. En la subexpresión `(Ord a) => [a] -> [a]`, `(Ord a)` *restringe* el tipo de datos sobre los que `a` puede variar. La utilización de este tipo de restricciones con frecuencia realza la calidad de un programa, al señalar con mayor precisión los tipos de los argumentos de una función que con la signatura simple.

9.2 El algoritmo de ordenamiento por fusión

El siguiente es un algoritmo que ordena una lista de números por fusión:

Material de Haskell: [9.2]

```
1 msort :: (Ord a) => [a] −> [a]
2 msort [] = []
```

```
msort [a] = [a]
msort (a:b:xs) = merge (msort xs1) (msort xs2)
                 where
                  xs1 = take k (a:b:xs)
                  xs2 = drop k (a:b:xs)
                  k  = div (length (a:b:xs)) 2

merge :: (Ord a) => [a] -> [a] -> [a]
merge [] bs = bs
merge as [] = as
merge as@(x:xs) bs@(y:ys)
        | (x<=y)   = x:(merge xs bs)
        | otherwise = y:(merge as ys)
```

9.3 El algoritmo de ordenamiento por selección

El siguiente algoritmo ordena una lista de números por selección:

Material de Haskell: [9.3]

```
ssort :: (Ord a) => [a] -> [a]
ssort [] = []
ssort (a:bs) = m: ssort (delete m (a:bs))
                        where m = minlist (a:bs)
delete a [] = []
delete a (x:bs) = if x==a then bs
                          else x:(delete a bs)
minlist [a] = a
minlist (a:b:xs) = if a<b then minlist (a:xs)
                          else minlist (b:xs)
```

10. CONCLUSIONES

La programación funcional tipo Haskell a veces parece misteriosa e innecesariamente orientada a las Matemáticas, y conocemos a más de uno que le dan pavor las Matemáticas. En Haskell, en particular, no existe asignamiento destructivo, y esto hace una de las características que Haskell sea *puro*. La ganancia que obtenemos de este hecho es que podemos tener la seguridad en alto porcentaje de que nuestros programas harán lo que deben hacer. Si el lector se anima a aprender más de Haskell, y a parecer todo un mago de la programación sin todavía saber mucho de computadoras, puede empezar a basarse en la siguiente bibliografía comentada.

Bibliografía comentada

Un excelente texto de programación funcional es [BW88]. En la segunda edición del libro, [Bir98], existe material específico a Haskell, además de una explicación práctica del concepto de *mónada*, una herramienta conceptual para considerar declarativamente a conceptos fuertemente imperativos, tales como los cambios de estado o la salida y entrada de programas. El artículo que replanteó la necesidad de utilizar herramientas alternativas a las existentes (en su tiempo) es [Bac78].

El tutorial por excelencia del lenguaje *Clean*, otro lenguaje funcional puro, es [KPvES01]. De hecho, ahí podrás encontrar hasta un juego de un pato con gráficas sorprendes y música integrada. Un lenguaje fuertemente tipeado es *Caml*. En el libro [CM98] hay una exposición del Caml, además de aquí se enfatiza en la parte *metodológica* para programar funcionalmente. Ocaml, una implementación específica de Caml, te sorprenderá en la cantidad de ejemplos que su distribución trae.

Como muchos de los títulos en inglés que involucran "craft" (oficio, en español), el libro [Tho90] es una obra valiosísima para poner manos a la obra en la programación utilizando Haskell.

Finalmente, el libro [Hud00] presenta a la Programación Funcional utilizando Haskell y un tema típicamente imperativo: las aplicaciones multimedia. Los programas aquí presentados funcionan bien con el intérprete *Hugs*[1]. Un primo cercano a Haskell con el que podrías experimentar tus incursiones al mundo funcional es *Scheme*[2], otro lenguaje de programación funcional un poco más tradicional en su concepción.

Agradecimientos

Un agradecimiento especial para Luis Zarza por su invitación en la redacción de este artículo.

11. REFERENCIAS

[Bac78] John Backus. Can programming be liberated from the von Newmann style? A functional style and its algebra of programs. *Communications of the Association for Computing Machinery*, 8(21):613–641, agosto 1978.

[Bir98] Richard Bird. *An introduction to functional programming using Haskell*. Prentice–Hall, second edition, 1998.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Series in Computer Science. Prentice Hall, 1988.

[CM98] Guy Cousineau and Michel Mauny. *The functional approach to programming*. Cambridge University Press, 1998.

[Dav92] Antony J.T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge Computer Science Texts. Cambridge University Press, 1992.

[Hud00] Paul Hudak. *The Haskell School of Expression. Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[KPvES01] Pieter Koopman, Rinus Plasmeijer, Marko van Eekelen, and Sjaak Smetsers. *Functional Programming in Clean*. `www.cs.kun.nl/~clean`, 2001.

[Mac90] Bruce J. MacLennan. *Functional programming. Practice and theory*. Addison–Wesley, 1990.

[Tho90] Simon Thompson. *Haskell: The craft of Functional Programming*. Addison–Wesley, second edition, 1990.

[1] Ver en *Internet* `http://www.haskell.org`.

[2] Ir en *Internet* a `http://www.plt-scheme.org`