Álgebra constructiva en Haskell

Facultad de Matemáticas
Departamento de Ciencias de la Computación e Inteligencia Artificial
Trabajo Fin de Grado

Autor

Agradecimientos

El presente Trabajo Fin de Grado se ha realizado en el Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla.

Supervisado por

Tutor

Abstract

Resumen en inglés

Índice general

Capítulo 1

Programación funcional con Haskell

En este capítulo se hace una breve introducción a la programación funcional en Haskell suficiente para entender su aplicación en los siguientes capítulos. Para una introducción más amplia se pueden consultar los apuntes de la asignatura de Informática de 1º del Grado en Matemáticas ([1]).

El contenido de este capítulo se encuentra en el módulo `PFH`

```
module PFH where
import Data.List
```

1.1. Introducción a Haskell

En esta sección se introducirán funciones básicas para la programación en Haskell. Como método didáctico, se empleará la definición de funciones ejemplos, así como la redefinición de funciones que Haskell ya tiene predefinidas, con el objetivo de que el lector aprenda "*a montar en bici, montando*".

A continuación se muestra la definición `(cuadrado x)` es el cuadrado de `x`. Por ejemplo, La definición es

```
-- |
-- >>> cuadrado 3
-- 9
-- >>> cuadrado 4
-- 16
cuadrado :: Int -> Int
cuadrado x = x * x
```

A continuación se muestra la definición `(cubo x)` es el cuadrado de `x`. Por ejemplo, La definición es

```
-- |
-- >>> cubo 3
-- 27
-- >>> cubo 2
-- 8
-- >>> cubo 4
-- 64
cubo :: Int -> Int
cubo x = x^3
```

S continuación se muestra la definición `(suma x y)` es la suma de `x` e `y`. Por ejemplo, La definición es

```
-- |
-- >>> suma 3 5
-- 8
-- >>> suma 4 2
-- 6
suma :: Int -> Int -> Int
suma x y = x + y
```

.

Capítulo 2

Como empezar con Emacs en Ubuntu

En este capítulo se hace una breve explicación de conceptos básicos para empezar a redactar un documento a LaTeX en Emacs y con Haskell a la vez, así como ir actualizando los archivos junto con la plataforma Github. Comenzaremos explicando como realizar la instalación de Ubuntu 16.04 en un PC con windows 10.

2.0.1. Instalar Ubuntu 16.04 junto a windows 10

Para realizar la instalación de Ubuntu junto a windows necesitaremos los siguientes programas:

+ Rufus-2.17

+ Ubuntu 16.04

También necesitaremos un pen drive para usarlo de instalador.

Paso 1:
Descargamos Ubuntu 16.04 y rufus-2.17 desde sus respectivas web (o enlaces dados anteriormente).
Necesitamos saber que tipo tiene nuestro disco duro, esto lo podemos ver haciendo click derecho sobre el icono de windows (abajo izquierda) y le damos a administrador de equipos ->administrador de discos, y nos aparecerá nuestro disco duro con todas sus subparticiones internas, en el general nos pondrá si es NTFS o MBR.

Paso 2:
Conectamos el pen al PC y abrimos el programa rufus, el propio programa reconocerá el pen, sino en la pestaña de dispositivo marcamos el pen.

En Tipo de partición si nuestro disco es NTFS marcamos GPT para UEFI, en caso contrario uno de los otros dos MBR.

En la parte de opciones de formateo marcamos (aunque deben de venir marcadas):

- Formateo rápido

- Crear disco de arranque con ->seleccionamos imagen ISO y con hacemos click en el icono de la derecha para adjuntar la imagen ISO de Ubuntu que hemos descargado anteriormente.

- Añadir etiquetas extendidas e iconos.

Y le damos a empezar.

Paso 3:
Dejamos el pen conectado al PC y reiniciamos el ordenador, al reiniciar justo antes de que cargue pulsamos F2 (o F1 según el PC) para acceder a la bios del PC y aqui nos vamos a la zona de arranque de cada sistema (esto cada bios es diferente) y tenemos que colocar el pen en la primera posición que en esta debe estar windows de esta forma iniciamos con el pen y comenzamos a instalar Ubuntu, seguimos los pasos solo tenemos que marcar España cuando nos lo pida y dar el espacio que queramos a Ubuntu con unos 40 GB sobra, el propio Ubuntu se encarga de hacer la partición del disco duro.

Paso 4:
Una vez instalado Ubuntu, nos vamos al icono naranjita que se llama software de Ubuntu y actualizamos.

Tras realizar todos estos pasos, cuando iniciemos el PC nos debe dar a elegir entre iniciar con Ubuntu o con Windows 10. Recomiendo buscar en youtube un video tuto-

rial de como instalar Ubuntu junto a windows 10.

2.0.2. Iniciar un Capítulo

Paso 1:
Abrimos el directorio desde Emacs con Ctrl+x+d y accedemos a la carpeta de texto para crear el archivo nuevo .tex sin espacios.

Paso 2:
Hacemos lo mismo pero en la carpeta código y guardamos el archivo con la abreviatura que hemos usado en el .tex, el archivo lo guardamos como .lhs para tener ahí el código necesario de Haskell.

Paso 3:
Al acabar el capitulo hay que actualizar el trabajo para que se quede guardado, para ello nos vamos a archivo que contiene todo el trabajo que en nuestro caso se llama 'TFG.tex' importante coger el de la extensión .tex, nos vamos a la zona donde incluimos los capitulos y usamos el comando de LaTeX con el nombre que le dimos en la carpeta de texto:

```
include{'nombre sin el .tex'}
```

2.0.3. Abreviaciones de Emacs:

La tecla ctrl se denominara C y la tecla alt M, son las teclas mas utilizadas, pues bien ahora explicamos los atajos más importantes y seguiremos la misma nomenclatura de la guía para las teclas:

ctrl es llamada C y alt M.

Para abrir o crear un archivo:
C + x + C + f

Para guardar un archivo:
C + x + C + s

Para guardar un archivo (guardar como):
C + x + C + w

Si abriste mas de un archivo puedes recorrerlos diferentes buffers con
C + x + ← o →

Emacs se divide y maneja en buffers y puedes ver varios buffers a la vez (los buffers son como una especie de ventanas).

Para tener 2 buffers horizontales:
C + x + 2

Para tener 2 buffers verticales (si hacen estas combinaciones de teclas seguidas verán que los buffers se suman):
C + x + 3

Para cambiar el puntero a otro buffer:
C + x + o

Para tener un solo buffer:
C + x + 1

Para cerrar un buffer:
C + x + k

Si por ejemplo nos equivocamos en un atajo podemos cancelarlo con:
C + g

Para cerrar emacs basta con:
C + x + C + C

Para suspenderlo:
C + z

Podemos quitar la suspensión por su id que encontraremos ejecutando el comando:

jobs

Y después ejecutando el siguiente comando con el id de emacs:

fg

Escribimos shell y damos enter.

2.0.4. Push and Pull de Github con Emacs

Vamos a mostrar como subir y actualizar los archivos en la web de Github desde la Consola (o Terminal), una vez configurado el pc de forma que guarde nuestro usuario y contraseña de Github. Lo primero que debemos hacer es abrir la Consola:

Ctrl+Alt+T

Escribimos los siguientes comandos en orden para subir los archivos:

cd 'directorio de la carpeta en la que se encuentran las subcarpetas de código y texto'
ejemplo: cd Escritorio/AlgebraConEnHaskell/

git add . (de esta forma seleccionamos todo)

git commit -m 'nombre del cambio que hemos hecho'

git push origin master

Para descargar los archivos hacemos lo mismo cambiando el último paso por:

```
git pull origin master
```

El contenido de este capítulo se encuentra en el módulo `ICH`

```
module ICH where
import Data.List
```

.

Capítulo 3

Teoría de anillos en Haskell

En este capítulo se muestra cómo definir la teoría de anillos en Haskell. Los anillos se pueden definir de forma compacta en grupos y monoides, pero daremos unas series de definiciones que los definen de forma más rigurosa.

Comenzamos dando las primeras definiciones y propiedades básicas que tiene un anillo en el módulo `TAH`

```
module TAH
   ( Ring(..)
   , propAddAssoc, propAddIdentity, propAddInv, propAddComm
   , propMulAssoc, propMulIdentity, propRightDist, propLeftDist
   , propRing
   , (<->)
   , sumRing, productRing
   , (<^>), (~~), (<**)
   ) where

import Data.List
import Test.QuickCheck
```

3.1. Teoría de anillos en Haskell

En esta sección, que corresponde con el fichero TAH.lhs, introducimos los conceptos básicos de la Teoría de Anillos. El objetivo es definir los conceptos mediante programación funcional y teoría de tipos.

En primer lugar, damos la definición teórica de anillos:

Definición 1. *Un anillo es una terna* $(R, +, *)$, *dondeResunconjuntoy+, *sondosoperacionesbinarias+,*:R R, (llamadas d usualmente suma y multiplicación) verificando las siguientes propiedades:*

1. *Asociatividad de la suma:* $\forall\ a, b, c\ \in\ R.\ \ (a + b) + c = a + (b + c)$

2. *Existencia del elemento neutro para la suma:* $\exists\ 0\ \in\ R.\ \forall\ a\ \in\ R.\ \ 0 + a = a + 0 = a$

3. *Existencia del inverso para la suma:* $\forall\ a\ \in\ R, \exists\ b\ \in\ R.\ \ a + b = b + a = 0$

4. *La suma es commutativa:* $\forall\ a, b\ \in\ R.\ \ a + b = b + a$

5. *Asociatividad de la multiplicación:* $\forall\ a, b, c\ \in\ R.\ \ (a * b) * c = a * (b * c)$

6. *Existencia del elemento neutro para la multiplicación:*

$$\exists\ 1\ \in\ R.\ \ \forall\ a\ \in\ R.\ \ \ 1 * a = a * 1 = a$$

7. *Propiedad distributiva a la izquierda de la multiplicación sobre la suma:*

$$\forall\ a, b, c\ \in\ R.\ \ a * (b + c) = (a * b) + (a * c)$$

8. *Propiedad distributiva a la derecha de la multiplicación sobre la suma:*

$$\forall\ a, b, c\ \in\ R.\ \ (a + b) * c = (a * c) + (b * c)$$

Representaremos la noción de anillo en Haskell mediante una clase. Para ello, declaramos la clase Ring sobre un tipo a con las operaciones internas que denotaremos con los símbolos $< + >$ y $< ** >$ (nótese que de esta forma no coinciden con ninguna operación previamente definida en Haskell). Representamos el elemento neutro de la suma mediante la constante *zero* y el de la multiplicación mediante la constante *one*. Asímismo, mediante la operacion *neg* representamos el elemento inverso para la suma.

```
infixl 6 <+>
infixl 7 <**>

class Ring a where
   (<+>) :: a -> a -> a
   (<**>) :: a -> a -> a
   neg :: a -> a
   zero :: a
   one :: a
```

```
-- |1. Asociatividad de la suma.
propAddAssoc :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propAddAssoc a b c = ((a <+> b) <+> c == a <+> (b <+> c), "propAddAssoc")
-- |2. Existencia del elemento neutro para la suma.
propAddIdentity :: (Ring a, Eq a) => a -> (Bool,String)
propAddIdentity a = (a <+> zero == a && zero <+> a == a, "propAddIdentity")
-- |3. Existencia del inverso para la suma.
propAddInv :: (Ring a, Eq a) => a -> (Bool,String)
propAddInv a = (neg a <+> a == zero && a <+> neg a == zero, "propAddInv")
-- |4. La suma es commutativa.
propAddComm :: (Ring a, Eq a) => a -> a -> (Bool,String)
propAddComm x y = (x <+> y == y <+> x, "propAddComm")
-- |5. Asociatividad de la multiplicación.
propMulAssoc :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propMulAssoc a b c = ((a <**> b) <**> c == a <**> (b <**> c), "propMulAssoc")
-- |6. Existencia del elemento neutro para la multiplicación.
propMulIdentity :: (Ring a, Eq a) => a -> (Bool,String)
propMulIdentity a = (one <**> a == a && a <**> one == a, "propMulIdentity")
-- |7. Propiedad distributiva a la izquierda de la multiplicación sobre la suma.
propRightDist :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propRightDist a b c =
  ((a <+> b) <**> c == (a <**> c) <+> (b <**> c), "propRightDist")
-- |8. Propiedad distributiva a la derecha de la multiplicación sobre la suma.
propLeftDist :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propLeftDist a b c =
 (a <**> (b <+> c) == (a <**> b) <+> (a <**> c), "propLeftDist")
```

Para saber si un conjunto a es un anillo se necesita una función que verifique las propiedades correspondientes:

```
-- | Test para ver que las propiedades satisfacen los axiomas de los anillos.
propRing :: (Ring a, Eq a) => a -> a -> a -> Property
propRing a b c = whenFail (print errorMsg) cond
  where
  (cond,errorMsg) =
    propAddAssoc a b c &&& propAddIdentity a  &&& propAddInv a        &&&
    propAddComm a b    &&& propMulAssoc a b c &&& propRightDist a b c &&&
    propLeftDist a b c &&& propMulIdentity a
  (False,x) &&& _         = (False,x)
  _         &&& (False,x) = (False,x)
  _         &&& _         = (True,"")
```

Veamos algunos ejemplos de anillos. Para ello, mediante instancias, especificamos las operaciones que dotan al conjunto de estructura de anillo. Por ejemplo, el anillo de los números enteros $\mathbb{Z}$ (en Haskell es el tipo *Integer*), con la suma y la multiplicación. Ejemplo:

```
-- | El anillo de los enteros con la operaciones usuales:
instance Ring Integer where
     (<+>)  = (+)
     (<**>) = (*)
     neg    = negate
     zero   = 0
     one    = 1
```

Veamos ahora cómo definir nuevas operaciones en un anillo a partir de las propias del anillo. En particular, vamos a definir la diferencia, la potencia, etc. Estas operaciones se heredan a las instancias de la clase anillo y, por tanto, no habría que volver a definirlas para cada anillo particular.

En primer lugar, establecemos el orden de prioridad para los símbolos que vamos a utilizar para denotar las operaciones.

```
infixl 8 <^>
infixl 6 <->
infixl 4 ~~
infixl 7 <**
```

```
-- | Diferencia.
(<->) :: Ring a => a -> a -> a
a <-> b = a <+> neg b
-- | Suma de una lista de elementos.
sumRing :: Ring a => [a] -> a
sumRing = foldr (<+>) zero
-- | Producto de una lista de elementos.
productRing :: Ring a => [a] -> a
productRing = foldr (<**>) one
-- | Potencia.
(<^>) :: Ring a => a -> Integer -> a
x <^> 0 = one
x <^> y | y < 0     = error "<^>: Input should be positive"
        | otherwise = x <**> x <^> (y-1)
-- | Relación de semiigualdad: dos elementos son semiiguales si son
--   iguales salvo el signo.
(~~) :: (Ring a, Eq a) => a -> a -> Bool
x ~~ y = x == y || neg x == y || x == neg y || neg x == neg y
```

Finalmente definimos la multiplicación de un entero por la derecha, la multiplicación de un entero por la izquierda se tiene debido a que la operación <+>es commutativa.

```
-- |Multiplicación de un entero por la derecha.
(<**) :: Ring a => a -> Integer -> a
_ <** 0 = zero
x <** n | n > 0     = x <+> x <** (n-1)
        | otherwise = neg (x <** abs n) -- error "<**: Negative input"
```

Para describir los anillos commutativos necesitamos un nuevo módulo `TAHCommutative`

```
module TAHCommutative
   (module TAH
   , CommutRing(..)
   , propMulComm, propCommutRing
   ) where

import Test.QuickCheck
import TAH
```

En este módulo introducimos el concepto de anillo conmutativo, que visto desde el punto de vista de la programación funcional, es una subclase de la clase *Ring*. Solo necesitaremos una función para definirlo, damos primero su definición teórica.

Definición 2. *un anillo conmutativo es un anillo (R, +, *) con elemento unidad, el elemento neutro, en el que la operación de multiplicación * es conmutativa; es decir,*
$\forall\ a, b\ \in\ R.\ a * b = b * a$

```
class Ring a => CommutRing a
propMulComm :: (CommutRing a, Eq a) => a -> a -> Bool
propMulComm a b = a <**> b == b <**> a
```

Para saber si un anillo es commutativo se necesita una función que verifique la propiedad:

```
-- | Test que comprueba si un anillo es commutativo.
propCommutRing :: (CommutRing a, Eq a) => a -> a -> a -> Property
propCommutRing a b c = if propMulComm a b
                               then propRing a b c
                               else whenFail (print "propMulComm") False
```

Dentro de este último módulo podemos definir el dominio de integridad, pues se necesita que el anillo sea conmutativo, usaremos el módulo `TAHIntegralDomain`

```
module TAHIntegralDomain
```

```
   (module TAHCommutative
   , IntegralDomain
   , propZeroDivisors, propIntegralDomain
   ) where

import Test.QuickCheck
import TAH
import TAHCommutative
```

Definición 3. *Dado un anillo A, un elemento $a \in A$ se dice que es un divisor de cero si existe $b \in A - \{0\}$ tal que $a * b = 0$. Un anillo A se dice dominio de integridad, si el único divisor de cero es* 0.
$\forall\ a, b \in R.\ a * b = 0\ a = 0\ or\ b = 0$

```
-- | Definición de dominios integrales.
class CommutRing a => IntegralDomain a
-- | Un dominio integral es un anillo que
propZeroDivisors :: (IntegralDomain a, Eq a) => a -> a -> Bool
propZeroDivisors a b = if a <**> b == zero then
                              a == zero || b == zero else True
```

Para saber si un anillo es un dominio de integridad usaremos la siguiente función:

```
propIntegralDomain :: (IntegralDomain a, Eq a) => a -> a -> a -> Property
propIntegralDomain a b c = if propZeroDivisors a b
                              then propCommutRing a b c
                              else whenFail (print "propZeroDivisors") False
```

Utilizaremos un nuevo módulo para dar la definición de cuerpo que se encuentra en `TAHCuerpo`

```
module TAHCuerpo
  ( module TAHIntegralDomain
  , Field(inv)
  , propMulInv, propField
  , (</>)
  ) where

import Test.QuickCheck

import TAH
import TAHIntegralDomain
```

Definición 4. *Un cuerpo es un anillo conmutativo con elemento unidad tal $(A - \{0\})$ también es un grupo abeliano, es decir, cumple las 4 primeras propiedades de un anillo.*

```
-- | Definición de cuerpo.
class IntegralDomain a => Field a where
  inv :: a -> a
propMulInv :: (Field a, Eq a) => a -> Bool
propMulInv a = a == zero || inv a <**> a == one
```

Para saber si un anillo conmutativo es un cuerpo usaremos la función:

```
propField :: (Field a, Eq a) => a -> a -> a -> Property
propField a b c = if propMulInv a
                  then propIntegralDomain a b c
                  else whenFail (print "propMulInv") False
```

Los cuerpos poseen otras operaciones además de las que un anillo conmutativo pueda tener, como es la división. Para poder dar dicha definición establecemos el orden de prioridad para el símbolo de la división.

```
infixl 7 </>

-- | División
(</>) :: Field a => a -> a -> a
x </> y = x <**> inv y
```

Bibliografía

[1] J. Alonso. Temas de programación funcional. Technical report, Univ. de Sevilla, 2015.

Índice alfabético