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.

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

```
module TAH where
import Data.List
```

3.1. Anillos en Haskell

En Haskell con la palabra 'data' podemos definir un nuevo tipo de clase dato. Ejemplo: "data Bool = False | True", La parte a la izquierda del = denota el tipo y la parte derecha son los constructores de datos qu especifican los diferentes valores que puede tener un tipo. En esta sección crearemos la clase de los Anillos (veáse que usaremos $**$ para la multiplicación definida anteriormente por $*$):

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

De esta forma definimos la clase Ring con sus constructores. Explicamos como funciona cada constructor:

(<+>) :: a ->a ->a y (<**>) :: a ->a ->a , la operación suma es <+>y la multiplicación <**>(aquí ponemos dos * porque está definida la opración en Haskell con <*>) recibe dos elementos y devuelve uno. Al poner .ª"no especificamos un tipo concreto.
neg recibe un elemento a y devuelve otro elemento, zero y one es un elemento predefinido de la clase.

De esta forma podemos definir funciones dentro de la clase Ring, y usará las operaciones definidas en los constructores. Así podemos construir las propiedades que definen a un anillo en Haskell.

La clase Ring será una instancia pues será una clase que pertenecerá a otra clase más grande. Pues un tipo puede ser una instancia de una clase si soporta ese comportamiento, es decir trabaja con las mismas operaciones o bien es un "subconjunto"que parte de esa clase.

Definición 1. *Un anillo es un conjunto R definido por dos operaciones binarias llamadas suma y multiplicación denotadas $+, * : R \times R \to R$ respectivamente.*
Los axiomas de la terna $(R, +, *)$ deben satisfacer:

1. *Cerrado para la suma:* $\forall\ a, b \in R.\ \ a + b \in R$

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

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

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

```
-- |2. Addition is associative.
propAddAssoc :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propAddAssoc a b c = ((a <+> b) <+> c == a <+> (b <+> c), "propAddAssoc")

-- |3. Zero is the additive identity.
propAddIdentity :: (Ring a, Eq a) => a -> (Bool,String)
propAddIdentity a = (a <+> zero == a && zero <+> a == a, "propAddIdentity")

-- |4. Negation give the additive inverse.
propAddInv :: (Ring a, Eq a) => a -> (Bool,String)
propAddInv a = (neg a <+> a == zero && a <+> neg a == zero, "propAddInv")
```

---

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

```
-- |5. Addition is commutative.
propAddComm :: (Ring a, Eq a) => a -> a -> (Bool,String)
propAddComm x y = (x <+> y == y <+> x, "propAddComm")
```

6. Cerrado bajo la multiplicación: $\forall\ a, b\ \in\ R.\ \ a * b \in R$

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

8. Existencia del elemento neutro para la multiplicación:

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

9. Propiedad distributiva a la izquierda de la multiplicación sobre la suma:

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

10. Propiedad distributiva a la derecha de la multiplicación sobre la suma:

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

```
-- |7. Multiplication is associative.
propMulAssoc :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propMulAssoc a b c = ((a <**> b) <**> c == a <**> (b <**> c), "propMulAssoc")

-- |8. One is the multiplicative identity.
propMulIdentity :: (Ring a, Eq a) => a -> (Bool,String)
propMulIdentity a = (one <**> a == a && a <**> one == a, "propMulIdentity")

-- |9. Multiplication is right-distributive over addition.
propRightDist :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propRightDist a b c =
  ((a <+> b) <**> c == (a <**> c) <+> (b <**> c), "propRightDist")

-- |10. Multiplication is left-ditributive over addition.
propLeftDist :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propLeftDist a b c =
 (a <**> (b <+> c) == (a <**> b) <+> (a <**> c), "propLeftDist")
```

Aquí definimos las primeras propiedades,
propAddAssoc está definida sobre la clase Ring y la clase Eq (equivalencias) toma 3 valores de entrada del tipo a y devuelve True o False (de aquí el (Bool,String) String es cadena de

caracteres y Bool el tipo de booleanos que devuelve True o False.
propAddIdentity y propAddInv está definidas de igual forma y toman un elemento del tipo a y devuelve True o False.
propAddComm a diferencia de los anteriores solo recibe dos elementos y devuelve el True o False. El resto de propiedades están definidas de la misma forma que las descritas anteriormente.

Si un anillo cuenta con un elemento neutro para la segunda operación se llama anillo unitario. A dicho elemento se le suele llamar la unidad (1) para diferenciarlo del elemento neutro de la primera operación (usualmente el 0).

El conjunto de los elementos no nulos de un anillo se escriben como R^*. Ejemplos de anillos como $(\mathbb{Z}, +, *)$ donde $+$ y $*$ denotan la suma y multiplicación ordinaria para los enteros. Otros ejemplos son $\mathbb{Q}, \mathbb{R}, \mathbb{C}$ con la definición usual de suma y multiplicación.

Los axiomas de los anillos también pueden ser representados en Haskell. Estos son representados como funciones que deben ser usadas para testear que una implementación satisface las condiciones:

```
-- | Specification of rings. Test that the arguments satisfy the ring axioms.
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,"")
```

Definimos la propiedad propRing definida en la clase Ring y Eq que recibe 3 elementos y lo que devuelve es Property (es un tipo de QuickCheckque se usa para comprobar). Normalmente, una Property es una función que devuelve un Booleano y comprobación como QuickCheck.

Ejemplo:

```
enterosZRing :: (Ring Int, Eq Int) => Int -> Int -> Int -> Property
enterosZRing 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,"")
```

Solo consideraremos los anillos commutativos, todos los ejemplos que daremos serán de anillos commutativos. Una clase fundamental de anillos finitos son el anillo de los enteros en modulo n, denotado por $\mathbb{Z}_n$. Esto se corresponde con los elementos $a \in \mathbb{Z}$ en la misma clase de congruencias modulo n, por ejemplo $\mathbb{Z}_3 \simeq \{0,1,2\}$ hay tres clases de congruencias modulo 3. La suma y multiplicación son definidas usando la suma y multiplicación de $\mathbb{Z}$ en modulo n.

El compilador de Haskell debería distinguir elementos de diferentes anillos y verificar que no se dupliquen. Por ejemplo el tipo de clase de $\mathbb{Z}$ depende de los valores de n. Es posible representar enteros segun la clase de Haskell pero es un poco difícil y dependemos de las clases que queremos tener.

Definición 2. *Un dominio integral es un anillo conmutativo satisfaciendo:*

$$a * b = 0 \;\Rightarrow\; a = 0 \vee b = 0, \quad \forall\, a, b \in R.$$

Los anillos $\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$ son dominios integrales con la definición de suma y multiplicación. Para $\mathbb{Z}$ es un poco más complicado, por ejemplo $\mathbb{Z}$ no es un dominio integral ya que $2 * 3 = 0$ modulo 6.Por tanto $\mathbb{Z}$ es un dominio integral si y solo n es primo. .

Bibliografía

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

Índice alfabético