

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

Esta obra está bajo una licencia Reconocimiento–NoComercial–CompartirIgual 2.5 Spain de Creative Commons.

Se permite:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:



Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor.



No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

Esto es un resumen del texto legal (la licencia completa). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

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 ([?]).

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. Recomendando 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
import Test.QuickCheck

--infixl 8 <^>
infixl 7 <*>
infixl 6 <+>
--infixl 6 <->
```

3.1. Anillos en Haskell

Comenzamos por definir los anillos en Haskell. Crearemos un modulo llamado TAH que contendrá todas las funciones necesarias para crear un anillo, así como sus principales propiedades. Primero daremos la definición teórica de anillos:

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

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 conmutativa:* $\forall a, b \in R. a + b = b + a$
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:*

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

Para dar la definición de anillos en Haskell usaremos las clases, declaramos la clase *Ring a* como la clase de los anillos donde *Ring* es el nombre de la clase para poder usarla sobre cualquier anillo *a*. Daremos las operaciones internas que necesita un anillo para ser definido, que son la suma (`<+>`), multiplicación (`<**>`), los elementos neutros para la suma (*zero*) y multiplicación (*one*) y el elemento inverso para la suma (*neg*). Para definir bien las operaciones que hemos usado al principio el comando *infixl*. Una vez construida la clase de los anillos, construimos las funciones de cada una de las propiedades que debe de cumplir *a* para ser un anillo.

```
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 conmutativa.
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 o no necesitaremos una función que compruebe que el conjunto dado verifique las propiedades dadas anteriormente:

```

-- | 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,"")

```

Veamos unos ejemplos de conjuntos que son anillos, para ello usaremos las instancias, de esta forma damos las operaciones que el conjunto tiene asociado para poder ser un anillo. El conjunto de los números enteros \mathbb{Z} (en Haskell es el tipo *Integer*), Ejemplo:

```
-- | El conjunto de los enteros.
instance Ring Integer where
    (<+>) = (+)
    (<**>) = (*)
    neg    = negate
    zero   = 0
    one    = 1
```

.

Bibliografía

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

Índice alfabético

cuadrado, 9

cubo, 10

suma, 10