, shadow , spanish]todonotes

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.

2.1. 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 tutorial de como instalar Ubuntu junto a windows 10.

2.2. 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'}
```

2.3. 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.

2.4. 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 daremos una breve introducción a los conceptos de teoría de anillos en Haskell para ello haremos uso de los módulos. Un módulo de Haskell es una colección de funciones, tipos y clases de tipos relacionadas entre sí. Un programa Haskell es una colección de módulos donde el módulo principal carga otros módulos y utiliza las funciones definidas en ellos para realizar algo. Así distribuiremos las secciones y partes del código que creeamos necesario en diferentes módulos. Nos centraremos principalmente en las notas sobre cómo definir los conceptos en la programación funcional y teoría de tipos.

3.1. Anillos

Comenzamos dando las primeras definiciones y propiedades básicas que tiene un anillo para posteriormente introducir los anillos conmutativos creamos el primer módulo `TAH`

Antes de empezar tenemos que crear nuestro módulo, todos tienen la misma estructura, se usa el comando de Haskell *module* seguido del nombre que le queramos dar al módulo. A continuación entre paréntesis introducimos todas las clases y funciones que vamos a definir y que queramos exportar cuando en otro fichero importemos este módulo, seguido del paréntesis escribimos *where* y finalmente importamos las librerías y módulos que vayamos a necesitar. Para importarlas usamos el comando *import*.

Para nuestro primer módulo solo usaremos la conocida librería de Haskell *Data.List* la cual comprende las operaciones con listas, y *Test.QuickCheck* esta librería contine las funciones para probar una propiedad e imprimir los resultados.

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

import Data.List
import Test.QuickCheck
```

Comenzamos con la parte teórica, damos la definición teórica de anillos:

Definición 1. *Un anillo es una terna $(R, +, *)$, donde R es un conjunto y $+, *$ son dos operaciones binarias $+, * : R \times R \to R$, (llamadas usualmente suma y multiplicación) verificando lo siguiente:*

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:*

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

Una vez tenemos la teoría, pasamos a implementarlo en Haskell. Representaremos la noción de anillo en Haskell mediante una clase. Para ello, declaramos la clase *Ring*

sobre un tipo *a* (es decir, *a* no está restringido a ningún otro tipo) 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 función *neg* representamos el elemento inverso para la suma, es decir, para cada elemento *x* del anillo, *negx* representará el inverso de *x* respecto de la suma $< + >$. Todas ellas varían según el anillo que queramos definir.

Para utilizar operaciones que definimos nosotros, (es decir, que no están implementadas en Haskell como puede ser la suma) usamos el operador de Haskell $infixl$. Los operadores $infix$ son en realidad funciones simples, y también se pueden definir mediante ecuaciones. Estos se usan en forma de "símbolos", a diferencia de los identificadores normales que son alfanuméricos. Se puede dar una declaración de fijeza para cualquier operador o constructor de $infix$, esta declaración especifica un nivel de precedencia de 0 a 9 (siendo 9 el más fuerte, se supone que la aplicación normal tiene un nivel de precedencia de 10), e izquierda, derecha o no asociatividad. La asociatividad izquierda se especifica vía $infixl$, esta la usaremos nosotros para introducir el símbolo de la operación que vamos a definir. La fijeza de más de un operador puede especificarse con la misma declaración de fijeza. Si no se proporciona una declaración de fijeza para un operador en particular, su valor predeterminado es $infixl$ 9. Por ello a la suma le daremos 6 y a la multiplicación 7 pues esta última tiene prioridad sobre la suma.

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

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

Una vez establecida la clase de los anillos, pasamos a implementar los axiomas de este. En Haskell un tipo es como una etiqueta que posee toda expresión. Esta etiqueta nos dice a que categoría de cosas se ajusta la expresión.

Todos los axiomas que tenemos que introducir tienen la misma estructura, recibe elementos que son del tipo *Ring* y del tipo *Eq* para devolver elementos del tipo *Bool* y

String.

La clase *Ring* la acabamos de definir y la clase de tipos *Eq* proporciona una interfaz para las comparaciones de igualdad. Cualquier tipo que tenga sentido comparar dos valores de ese tipo por igualdad debe ser miembro de la clase *Eq*. El tipo *Bool* devuelve un booleano con *True* y *False*, en nuestras funciones es necesario pues necesitamos que nos devuelva *True* si se verifica el axioma y *False* en caso contrario. El tipo *String* es sinónimo del tipo *Char* este es necesario pues los booleanos son una cadena de carácteres.

```
-- |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 una terna $(a, <+>, <**>)$ es un anillo crearemos una propiedad

que se encargue de comprobar que los axiomas anteriores se verifiquen, para cada caso particular de una instancia dada. La estructura que tiene es la siguiente: recibe un elemento de tipo *Ring* y *Eq* y devuelve un elemento de tipo Property. Este es un tipo que convierte lo que recibe en una propiedad, es una función importada desde el módulo *Test.QuickCheck*.

```
-- | Test para ver si se verifican los axiomas de un anillo.
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
```

Se admite esta instancia porque se ha comprobado que se verifican los axiomas para ser un anillo. En caso contrario, proporcionaría un error.

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 "<^>: La entrada debe ser positiva."
        | otherwise = x <**> x <^> (y-1)
-- | Relación de semi-igualdad: dos elementos son semi-iguales 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
```

En la función *sumRing* hemos usado el comando $foldrxc$, este se usa para aplicar la operación x a los elementos de una lista tomando como elemento de inicio a c.

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 conmutativa. Esta función al igual que la anterior de potencia recibe un elemento de tipo *Ring* y devuelve un número entero, que es el tipo *Integer*. Cuando lo que devuelve no tiene ningún tipo especificado significa que no tiene restricción de tipo.

```
-- |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 "<**: Entrada Negativa."
```

3.2. Anillos Conmutativos

Para espeficar los anillos commutativos crearemos un nuevo módulo en el que importaremos el módulo anterior, el nuevo módulo será `TAHCommutative`

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

import Test.QuickCheck
import TAH
```

En este módulo introducimos el concepto de anillo conmutativo. Aquí hemos importado el módulo *TAH* con el comando *import*, para poder usar las clases y funciones definidas en dicho módulo. Vito desde el punto de vista de la programación funcional,un anillo conmutativo 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* $(R, +, *)$ *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 crearemos una propiedad que compruebe, en cada caso particular, que las operaciones concretas de una instancia verifique los axiomas para ser un anillo conmutativo y por consiguiente un anillo :

```
-- | 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
```

3.3. Dominio de integridad y Cuerpos

Dada la noción de anillo conmutativo podemos hablar de estructuras algebraicas como dominio de integridad y cuerpo. Comenzamos por el módulo `TAHIntegralDomain`

```
module TAHIntegralDomain
   (module TAHCommutative
   , IntegralDomain
```

```
    , propZeroDivisors, propIntegralDomain
    ) where

import Test.QuickCheck
import TAHCommutative
```

Para iniciar este módulo necesitamos importar el módulo *TAHCommutative* ya que los dominios de integridad que vamos a utilizar son anillos conmutativos, por lo que la clase que vamos a definir parte del tipo *CommutRing* que como hemos definido antes es el tipo de los anillos conmutativos, damos su definición.

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*. Es decir,* $\forall\ a, b \in R.\ a * b = 0 \Rightarrow\ a = 0\ or\ b = 0$

```
-- | Definición de dominio de integridad.
class CommutRing a => IntegralDomain a
-- | Un dominio de integridad es un anillo cuyo único divisor de cero es 0.
propZeroDivisors :: (IntegralDomain a, Eq a) => a -> a -> Bool
propZeroDivisors a b = if a <**> b == zero then
                              a == zero || b == zero else True
```

Como ocurría con los axiomas de los anillos, la función *propZeroDivisors* requiere que los elementos que recibe sean de la clase de tipo *IntegralDomain* y de tipo *Eq* pues estamos definiendo operaciones en las que se tiene que dar una igualdad, y devuelva un valor booleano, por ello el elemento de salida es de tipo *Bool*.

Para determinar si un anillo es un dominio de integridad usaremos la siguiente propiedad, esta tal y como ocurre con las anteriores propiedades, se encarga de comprobar que para cualquier instancia que demos se cumplan los axiomas que tiene que verificar, en este caso, para ser un dominio de integridad:

```
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
```

Ahora podemos implementar las especificaciones de la noción de cuerpo en el módulo `TAHField`

```
module TAHField
  ( module TAHIntegralDomain
```

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

import Test.QuickCheck
import TAHIntegralDomain
```

Para poder implementar la noción de cuerpo, necesitamos importar el módulo anterior *TAHIntegralDomain*, pues si una terna $(A, +, *)$ es un cuerpo por consiguiente es dominio de integridad, y al definir la clase de cuerpo le imponemos la restricción de que sea un dominio de integridad, veamos la definición teórica de cuerpo.

Definición 4. *Un cuerpo es un anillo de división conmutativo, es decir, un anillo conmutativo y unitario en el que todo elemento distinto de cero es invertible respecto del producto. Otra forma de definirlo es la siguiente, un cuerpo R es un dominio de integridad tal que para cada elemento* $a \neq 0$*, existe un inverso* a^{-1} *que verifica la igualdad:* $a^{-1}a = 1$.

Esta segunda definición es la que usaremos para la implementación, por ser más constructiva. La primera definición es la más común a nivel de teoría algebraica, y para aquellos familiarizados con conceptos básicos de álgebra, conocen la definición de cuerpo como la primera que hemos dado.

En Haskell especificamos el inverso de cada elemento mediante la función *inv*. La función *propMulInv* esta restringida a la clase de tipo *Field* pues requerimos que sea cuerpo y al tipo *Eq* pues se tiene que dar la igualdad.

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

Especificamos la propiedad que han de verificar los ejemplos de cuerpos. Es decir, dada una terna $(A, +, *)$ para una instancia concreta, esta tiene que verificar los axiomas para ser un cuerpo.

```
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
```

En un cuerpo se puede definir 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
```

3.4. Ideales

En esta sección introduciremos uno de los conceptos importantes en el álgebra conmutativa, el concepto de ideal. Dado que solo consideramos anillos conmutativos, la propiedad multiplicativa de izquierda y derecha son la misma. Veamos su implementación en el módulo `TAHIdeal`

```
-- |Ideal finitamente generado en un anillo conmutativo.

module TAHIdeal
  ( Ideal(Id)
  , zeroIdeal, isPrincipal, fromId
  , addId, mulId
  , isSameIdeal, zeroIdealWitnesses
  ) where

import Data.List (intersperse,nub)
import Test.QuickCheck

import TAHCommutative
```

Para desarrollar está sección importamos el módulo *TAHConmutative*.

Definición 5. *Sea $(R, +, *)$ un anillo. Un ideal de R es un subconjunto $I \subset R$ tal que 1. $(I, +)$ es un subgrupo de $(R, +)$. 2. $RI \subset I$. Es decir, $\forall\ a \in A \forall\ b \in I, ab \in I$.*

La definición anterior da ideales arbitrarios de anillos y no es factible para el algebra constructiva. Enunciaremos otra definión de los ideales que nos proporcionará ideales finitamente generados, estos se pueden implementar en Haskell.

Definición 6. *Sea $(R, +, *)$ un anillo, y E un subconjunto de R. Se define el ideal generado*

por E, y se denota $< E >$, como la intersección de todos los ideales que contienen a E (que es una familia no vacía puesto que R es un ideal que contiene a E).

Se llama ideal generado por los elementos $e_1, .., e_r$ de un anillo $(A, +, *)$ al conjunto $E =< e_1, .., e_r >:= a_1e_1 + .. + a_re_r | a_1, ..a_r \in A$. Este conjunto es el ideal de A más pequeño que contiene a los elementos $e_1, .., e_r$. Cualquier elemento x del ideal generado por E, es una combinación lineal de los generadores. Es decir, si $x \in E$, existen coeficientes $\alpha_1, .., \alpha_r$ tales que $x = \alpha_1 x_1 + .. + \alpha_r x_r$.

Para el tipo de dato de los Ideales, en anteriores versiones de Haskell podiamos introducir una restricción al tipo que ibamos a definir mediante el constructor *data*, pero actualmente no se puede. El constructor *data* significa que vamos a definir un nuevo tipo de dato. La parte de la izquierda del $=$ denota el tipo y la parte de la derecha son los constructores de datos. Estos especifican los diferentes valores que puede tener un tipo.

Para especificar en Haskell el ideal generado por un conjunto finito E, con *data* crearemos el tipo de dato mediante el constructor Id y el conjunto E se representará por una lista de elementos del anillo. Por ejemplo, en el anillo de los enteros $\mathbb{Z}$, el ideal generado por $< 2, 5 >$ se representará por $Id[2, 5]$. Y el ideal canónico cero $< 0 >$ en cualquier anillo se representará por Id [zero], hay dos ideales canónico el cero ideal y todo el anillo R, este último se representará por $Id[one]$.

Sin embargo los ideales con los que trabajaremos están restringidos a anillos conmutativos. Para aplicar dicha restricción, lo haremos en cada definición de instancia o función, quedando explicito que usaremos los anillos conmutativos con la clase definida anteriormente como *CommutRing*.

```
-- |Ideales caracterizados por una lista de generadores.
data Ideal a = Id [a]

instance (CommutRing a, Show a) => Show (Ideal a) where
  show (Id xs) = "<" ++ concat (intersperse "," (map show xs)) ++ ">"

instance (CommutRing a, Arbitrary a, Eq a) => Arbitrary (Ideal a) where
  arbitrary = do xs' <- arbitrary
                 let xs = filter (/= zero) xs'
                 if xs == [] then return (Id [one]) else return (Id (nub xs))

-- | El ideal cero.
zeroIdeal :: CommutRing a => Ideal a
zeroIdeal = Id [zero]
```

Explicamos varios conceptos de Haskell utilizados en las anteriores líneas de código. Comenzamos por explicar las funciones utilizadas para declarar mediante una instancia a un ideal. La clase *Show* convierte el elemento que recibe en una cadena legible, es decir, cuando intentamos mostrar un valor por pantalla, primero Haskell ejecuta la función show para obtener la representación en texto de un dato y luego lo muestra en la terminal. Si añadimos $deriving(Show)$ al final de una declaración de tipo, automáticamente Haskell hace que ese tipo forme parte de la clase de tipos Show, y lo muestra como lo tenga por defecto. Mediante esta instancia modificamos esta presentación especificando como queremos que lo muestre. Por ejemplo, el ideal $Id[2,5]$ se va a mostrar como $< 2,5 >$.

La función *intersperse* toma un elemento y una lista, e intercala el elemento entre cada uno de los elementos de la lista. Por ejemplo: $intersperse\ ',\ "abcde" == "a,b,c,d,e"$. Con ambas funciones damos la forma de presentación del ideal, tal y como se representa de forma teórica.

Para la segunda instancia hemos utilizado la clase *Arbitrary*, esta proviene de la líbrería *QuickCheck*, proporciona una generación aleatoria y proporciona valores reducidos. Gracias a esta clase, con la segunda instancia podemos generar ideales de forma aleatoria. Esto será necesario para poder especificar propiedades sobre estos ideales y comprobarlas con QuickCheck.

En esta misma instancia se hace uso de la función $filter$, es una función que toma un predicado (un predicado es una función que dice si algo es cierto o falso, o en nuestro caso, una función que devuelve un valor booleano) y una lista y devuelve una lista con los elementos que satisfacen el predicado. La función $filter$ la utilizamos para eliminar los elementos nulos, si los hubiese, de la lista que toma.

Vamos a explicar brevemente como funciona la segunda instancia. Comienza generando una lista xs' de elementos cualesquiera del anillo, con $filter$ se filtra y se eliminan los ceros obteniendo la nueva lista xs. Si $xs = []$, se genera el ideal Id [one], todo el anillo; en caso contrario, el ideal generado por los elementos de xs, sin repeticiones (eliminadas con la función *nub*).

Tanto la función $filter$ como $foldr$ son funciones de orden superior. Las funciones de Haskell pueden tomar funciones como parámetros y devolver funciones como resultado. Una función que hace ambas cosas o alguna de ellas se llama función de orden

superior. Generación aleatoria y reducción de valores.

Finalmente hemos implementado uno de los ideales canónicos, el ideal cero, $< 0 >$. A continuación, damos la definición teórica de ideal principal.

Definición 7. *Un ideal $I \subset R$ se llama principal si se puede generar por un sólo elemento. Es decir, si $I =< a >$, para un cierto $a \in R$.*

Los anillos como Z en los cuales todos los ideales son principales se llaman clásicamente dominios de ideales principales. Pero constructivamente esta definición no es adecuada. Sin embargo, estamos considerando anillos conmutativos en los cuales todos los ideales son finitamente generados. Por tanto, estos son representados por un conjunto finito, y esto si podemos implementarlo a nivel computacional.

```
isPrincipal :: CommutRing a => Ideal a -> Bool
isPrincipal (Id xs) = length xs == 1
```

Mediante la función $fromId$, definida a continuación, mostramos el ideal en forma de lista.

```
fromId :: CommutRing a => Ideal a -> [a]
fromId (Id xs) = xs
```

Ahora veamos algunas operaciones sobre ideales y propiedades fundamentales de ideales, como pueden ser la suma y multiplicación. Por último daremos una función para identificar si dos ideales son el mismo ideal. Para realizar la implementación de estas operaciones, lo haremos solo para ideales finitamente generados.

Definición 8. *Si I y J son ideales de un anillo $(R, +, *)$, se llama suma de ideales al conjunto $I + J = a + b | a \in I, b \in J$.La suma de ideales también es un ideal.*

```
-- | Suma de ideales.
addId :: (CommutRing a, Eq a) => Ideal a -> Ideal a -> Ideal a
addId (Id xs) (Id ys) = Id (nub (xs ++ ys))
```

Definición 9. *Si I y J son ideales de un anillo $(R, +, *)$, se llama producto al conjunto $IJ = ab | a \in I, b \in J$. El producto de ideales también es un ideal.*

```
-- | Multiplicación de ideales.
mulId :: (CommutRing a, Eq a) => Ideal a -> Ideal a -> Ideal a
mulId (Id xs) (Id ys) = if zs == [] then zeroIdeal else Id zs
  where zs = nub [ f <**> g | f <- xs, g <- ys, f <**> g /= zero ]
```

A continuación veremos una función cuyo objetivo es es comprobar que el resultado de una operación 'op' sobre dos ideales calcula el ideal correcto. Para ello, la operación debería proporcionar un "testigo" de forma que el ideal calculado tenga los mismos elementos. Es decir, si z_k es un elemento del conjunto de generadores de $(Idzs)$, z_k tiene una expresión como combinación lineal de xs e ys, cuyos coeficientes vienen dados por as y bs respectivamente. respectivamente).

```
-- | Verificar si es correcta la operación entre ideales.

isSameIdeal :: (CommutRing a, Eq a)
            => (Ideal a -> Ideal a -> (Ideal a, [[a]], [[a]]))
            -> Ideal a
            -> Ideal a
            -> Bool
isSameIdeal op (Id xs) (Id ys) =
  let (Id zs, as, bs) = (Id xs) `op` (Id ys)
  in length as == length zs && length bs == length zs
     &&
     and [ z_k == sumRing (zipWith (<**>) a_k xs) && length a_k == length xs
         | (z_k,a_k) <- zip zs as ]
     &&
     and [ z_k == sumRing (zipWith (<**>) b_k ys) && length b_k == length ys
         | (z_k,b_k) <- zip zs bs ]
```

Explicamos con más detenimiento como funciona *isSameIdeal*, recibe como argumento una operación *op* que representa una operación entre los dos ideales que recibe. Es decir, la función *op* debería devolver una terna $(Idzs, as, bs)$, donde *as* y *bs* son listas de listas de coeficientes (justamente, los coeficientes de cada generador de *zs* en función de *xs* y de *ys*,respectivamente). La función *isSameIdeal* devuelve un booleano, si devuelve *True* nos indica que la operación que se ha realizado entre ambos ideales es correcta. Para ello, toma una terna formada por: (un ideal, lista de anillos conmutativos, lista de anillos conmutativos), seguido de un booleano sobre dos ideales. Realiza 3 comprobaciones, la primera comprueba que la longitud de las dos listas de la terna sean la misma que la del ideal de la terna. La segunda comprueba, mediante elementos de ambas listas tomadas a pares *zs* con *as*, que cada elemento del ideal *zs* sea el mismo que el elemento resultante al aplicar *sumRing* sobre el anillo generado con la función *zipWith* y además que la longitud de ambos coincida. La tercera es análoga a la anterior cambiando *as* por *bs*.
Es decir,

La función *zipWith* generaliza a la función *zip* comprimiendo con la función dada como primer argumento, en lugar de una función de tuplas.Por ejemplo, $zipWith(+)$ se aplica a dos listas para producir la lista de sumas correspondientes.

Para finalizar esta sección, implementamos la función zeroIdealWitnesses proporciona la función "testigo" para una operación sobre ideales cuyo resultado sea el ideal cero.

```
zeroIdealWitnesses :: (CommutRing a) => [a] -> [a] -> (Ideal a, [[a]], [[a]])
zeroIdealWitnesses xs ys = ( zeroIdeal
                           , [replicate (length xs) zero]
                           , [replicate (length ys) zero])
```

y seguidamente daremos las noción de un anillo que es especialmente relevante para el álgebra constructiva. Lo veremos implementado en el módulo `TAHStronlyDiscrete`

```
module TAHStronglyDiscrete
  ( StronglyDiscrete(member)
  , propStronglyDiscrete
  ) where

import TAHCommutative
import TAHIdeal
```

Para desarrolla esta pequeña sección, importamos los módulos TAHConmutative y TAHIdeal. Veamos antes unas definiciones teóricas.

Definición 10. *Un anillo se llama discreto si la igualdad es decidible.*

Todos los anillos consideremos serán discretos. Pero hay muchos ejemplos de anillos que no son discretos. Por ejemplo, R no es discreto ya que es no es posible decidir si dos números irracionales son iguales en tiempo finito.

Definición 11. *Un anillo es fuertemente discreto si podemos decidir que un elemento de un ideal es decidible.*

Esta propiedad es muy fuerte. Muchos de los anillos que hemos visto hasta ahora son muy discreto, esto de hecho está estrechamente relacionado con si la división es decidible en el anillo.

Para introducir este concepto crearemos una clase restringida a la clase de tipo *Ring*.

```
class Ring a => StronglyDiscrete a where
  member :: a -> Ideal a -> Maybe [a]
```

Para definir la función *member* hemos utilizado *Maybe*, un constructor de tipo. La *a* es un parámetro de tipo. Ningún valor puede tener un tipo que sea simplemente

Maybe, ya que eso no es un tipo por si mismo, es un constructor de tipos. Para que sea un tipo real que algún valor pueda tener, tiene que tener todos los parámetros de tipo definidos. De esta forma, con *Maybe* lo que estamos haciendo es decidir si el parámetro *a* es de tipo *Ideal* o no.

Damos a continuación la función para comprobar si un anillo conmutativo es fuertemente discreto.

```
propStronglyDiscrete :: (CommutRing a, StronglyDiscrete a, Eq a)
                     => a -> Ideal a -> Bool
propStronglyDiscrete x id@(Id xs) = case member x id of
  Just as -> x == sumRing (zipWith (<**>) xs as) && length xs == length as
  Nothing -> True
```

Explicamos brevemente como funciona *propStronglyDiscrete*. Esta recibe como argumentos un elemento x y $id@(Id\ xs)$ con @ lo que hacemos es como crear una función o guardar un valor en *id* de forma de que cuando llamemos a *id* nos estamos refiriendo a $Id\ xs$. En primer lugar con $case..of$ nos preguntamos si x pertenece al ideal generado por xs, es decir si x es un elemento del ideal. En caso de que sí pertenezca, debe de existir una lista *as* tal que el elemento x sea igual a la suma del anillo generado por la multiplicación de elementos a pares de xs y *as*, así como que ambas listas deben tener la misma longitud. Si esto se verifica devuelve *True* y nuestro anillo es fuertemente discreto.

Bibliografía

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

Índice alfabético