, 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:
$\boxed{\text{C + x + C + s}}$

Para guardar un archivo (guardar como):
$\boxed{\text{C + x + C + w}}$

Si abriste mas de un archivo puedes recorrerlos diferentes buffers con
$\boxed{\text{C + x + } \leftarrow \text{ o } \rightarrow}$

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:
$\boxed{\text{C + x + 2}}$

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

Para cambiar el puntero a otro buffer:
$\boxed{\text{C + x + o}}$

Para tener un solo buffer:
$\boxed{\text{C + x + 1}}$

Para cerrar un buffer:
$\boxed{\text{C + x + k}}$

Si por ejemplo nos equivocamos en un atajo podemos cancelarlo con:
$\boxed{\text{C + g}}$

Para cerrar emacs basta con:
$\boxed{\text{C + x + C + C}}$

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'

$\boxed{\text{git push origin master}}$

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

$\boxed{\text{git pull origin master}}$

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.

3.1. Anillos

Comenzamos dando las primeras definiciones y propiedades básicas que tiene un anillo para posteriormente introducir los anillos conmutativos 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
```

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 $(R, +, *)$, donde R es un conjunto y $+, *$ son dos operaciones binarias $+, * : R \times R \to R$, (llamadas 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.

```haskell
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 una terna $(a, <+>, <**>)$ es un anillo se necesita una función que verifique las propiedades correspondientes:

```
-- | Test para ver que se 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 "<^>: 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
```

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

3.2. Anillos Conmutativos

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* $(R, +, *)$ *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
```

3.3. Dominio de integridad y Cuerpos

Dadas las nociones de anillos conmutativos podemos introducir dos conceptos básicos dominio de integridad y cuerpos, comenzamos por el módulo `TAHIntegralDomain`

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

import Test.QuickCheck
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. Es decir, $\forall\ a, b \in R.\ a * b = 0 \Rightarrow a = 0$ 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 determinar 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
```

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

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

import Test.QuickCheck
import TAHIntegralDomain
```

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. Un cuerpo* R *es un dominio de integridad que contiene para cada elemento* $a \neq 0$ *un inverso* a^{-1} *que verifica la igualdad:* $a^{-1}a = 1$.

```
-- | 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
```

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

El concepto de ideales es muy importante en el álgebra conmutativa. Son generalizaciones de muchos conceptos de los enteros. Dado que solo consideramos anillos conmutativos, la propiedad multiplicativa de izquierda y derecha son la misma. Veamos su implementación en el módulo `TAHIdeal`

```
-- |Ideales finitamente generados en anillos conmutativos.

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

```
  ) where

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

import TAHCommutative
```

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,* $\forall\ a \in A \forall\ b \in I, ab \in I$*.*

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).*

Para el tipo de dato de los Ideales, en anteriores versiones de Haskell podiamos introducir una restricción al tipo que ibamos a definir con el comando *"data"*, pero actualmente no se puede. Sin embargo los ideales con los que trabajaremos están restringidos a anillos conmutativos. Para aplicar dicha restricción en cada definición de instancia o función, queda explicito que usaremos los anillos conmutativos con la clase definida anteriormente $CommutRing$.

```
-- |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]
```

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* $a \in R$*.*

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

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

```
-- |Evaluar un ideal en un cierto punto.
```

```
eval :: CommutRing a => a -> Ideal a -> a
eval x (Id xs) = foldr (<+>) zero (map (<**> x) xs)
```

La propiedad más importante de los ideales es que sirven para definir los anillos cocientes. Dado un ideal $I \subset R$, sabemos que $(I, +)$ es un subgrupo (abeliano) de $(R, +)$, y por tanto podemos considerar el grupo cociente A/I. Lo interesante es que en este grupo cociente, además de la suma: $(a + I) + (b + I) = (a + b) + I$,

```
-- | Addition of ideals.
addId :: (CommutRing a, Eq a) => Ideal a -> Ideal a -> Ideal a
addId (Id xs) (Id ys) = Id (nub (xs ++ ys))
```

Se puede definir un producto, de forma natural: $(a + I)(b + I) = ab + I$.

```
-- |  Multiplication of ideals.
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 ]
```

Este producto está bien definido porque I es un ideal. Además, la suma y el producto de clases de equivalencia que acabamos de definir, cumplen las propiedades necesarias que hacen de R/I un anillo: El anillo cociente de R sobre I.

Para determinar si dos ideales son el mismo usaremos la siguiente función:

```
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 ]
```

Daremos la función que genera dos listas de ideales que se completan con cero para calcular las intersecciones entre ideales.

```
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 nociones de unos anillos que son especialmente relevantes para la construcción matemática. Lo veremos implementado en el módulo `TAHStronlyDiscrete`

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

import TAHCommutative
import TAHIdeal
```

Definición 8. *Un anillo se llama discreto si la igualdad es decidible.*

Definición 9. *Un anillo es fuertemente discreto si podemos decidir que un elemento de un ideal es decidible, es decir, podemos decidir si un sistema $a_1x_1 + .. + a_nx_n = b$ tiene solución o no.*

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

Damos a continuación la función para comprobar si un elemento está en el ideal o no.

```
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
```

Capítulo 4

Anillos Coherentes

Todos los anillos en este capítulo son dominios integrales. Uno de los principales objetivos de las siguientes secciones serán demostrar que los diferentes anillos son coherentes. Esto significa que es posible resolver sistemas de ecuaciones en ello.

4.1. Matrices y Vectores en Haskell

Antes veremos una breve implementación de vectores y matrices en Haskell con el objetivo de conseguir resolver sistemas de ecuaciones con matrices del tipo $Ax = b$ mediante Gauss-Jordan, lo vemos en el módulo `TAHMatriz`

```haskell
-- | Una pequeña librería de matrices simples.
module TAHMatriz
  ( Vector(Vec)
  , unVec, lengthVec
  , Matrix(M), matrix
  , matrixToVector, vectorToMatrix, unMVec, unM, (!!!)
  , identity, propLeftIdentity, propRightIdentity
  , mulM, addM, transpose, isSquareMatrix, dimension
  , scale, swap, pivot
  , addRow, subRow, addCol, subCol
  , findPivot, forwardElim, gaussElim, gaussElimCorrect
  ) where

import qualified Data.List as L
import Data.Function (on)
import Control.Monad (liftM)
import Control.Arrow hiding ((<+>))
import Test.QuickCheck
```

```
import TAHCuerpo

import Debug.Trace
```

Comenzamos por implementar las nocines básicas de los vectores. Creamos un nuevo tipo para definir un vector, usaremos las listas para aplicarlas sobre el tipo *Vector* y trabajar con ellas como vectores en Haskell:

```
-- | Vectores.

newtype Vector r = Vec [r] deriving (Eq)

instance Show r => Show (Vector r) where
  show (Vec vs) = show vs

-- Generar un vector de longitud 1-10.
instance Arbitrary r => Arbitrary (Vector r) where
  arbitrary = do n <- choose (1,10) :: Gen Int
                 liftM Vec $ gen n
    where
    gen 0 = return []
    gen n = do x <- arbitrary
               xs <- gen (n-1)
               return (x:xs)

-- Aplicar una función f a un vector.
instance Functor Vector where
  fmap f = Vec . map f . unVec
```

Ahora vamos a definir la suma y multiplicación de vectores sobre un anillo. Estas operaciones tenemos que dejarlas comentadas pues para que funcionen tenemos que asignar el elemento neutro para la suma y producto, pues estamos definiendolo como una instancia de la clase *Ring*. Recordamos que para realizar operaciones con vectores estos tienen que tener la misma dimensión.

```
{-
instance Ring r => Ring (Vector r) where
  (Vec xs) <+> (Vec ys) | length xs == length ys = Vec (zipWith (<+>) xs ys)
                        | otherwise = error "Los vectores no se pueden sumar porque t
  (Vec xs) <**> (Vec ys) | length xs == length ys = Vec (zipWith (<**>) xs ys)
                         | otherwise = error "Los vectores no se pueden multiplicar p
  one  = ?
  zero = ?
-}
```

Para acabar con los vectores damos la función que muestra el vector y la que mide la longitud de un vector en ese formato.

```
unVec :: Vector r -> [r]
unVec (Vec vs) = vs

lengthVec :: Vector r -> Int
lengthVec = length . unVec
```

Una vez dadas las nocines de los vectores podemos comenzar a implementar las nociones de las matrices, notesé que cada fila o columna de una matriz puede verse como un vector. Haciendo uso del tipo *Vector* implementamos las matrices como nuevo tipo.

```
-- | Matrices
newtype Matrix r = M [Vector r]
  deriving (Eq)

instance Show r => Show (Matrix r) where
  show xs = case unlines (map show (unMVec xs)) of
    [] -> "[]"
    xs -> init xs ++ "\n"

-- Generar matrices con como mucho 10 filas.
instance Arbitrary r => Arbitrary (Matrix r) where
  arbitrary = do n <- choose (1,10) :: Gen Int
                 m <- choose (1,10) :: Gen Int
                 xs <- sequence [ liftM Vec (gen n) | _ <- [1..m]]
                 return (M xs)
    where
    gen 0 = return []
    gen n = do x <- arbitrary
               xs <- gen (n-1)
               return (x:xs)

-- aplicar una función f a una matriz.
instance Functor Matrix where
  fmap f = M . map (fmap f) . unM
```

Una vez implementado el tipo de las matrices vamos a crear la función para construir una matriz de dimensión mxn a partir de una lista de vectores, de forma que cada vector es una fila de la matriz (todos de la misma longitud) y la longitud un vector es el número de columnas.

```
-- | Matriz mxn.
```

```
matrix :: [[r]] -> Matrix r
matrix xs =
  let m = fromIntegral $ length xs
      n = fromIntegral $ length (head xs)
  in if length (filter (\x -> fromIntegral (length x) == n) xs) == length xs
        then M (map Vec xs)
        else error "La dimensión del vector no puede ser distinta al resto"
```

Las siguientes funciones son para mostrar una matriz como lista de vectores, y aplicar funciones con este formato sobre ella. Pasar de matrices a vectores y viceversa, así como una función para comprobar que las dimensiones de la matriz son correctas y ninguna es mayor de 10 (Notesé que para esta última función hemos utilizado el símbolo (!!!) que se utiliza en mátematicas para las contradicciones).

```
-- Mostrar la matriz como lista de vectores.
unM :: Matrix r -> [Vector r]
unM (M xs) = xs

-- Aplicar lo anterior a cada vector de la lista.
unMVec :: Matrix r -> [[r]]
unMVec = map unVec . unM

-- De vector a matriz.
vectorToMatrix :: Vector r -> Matrix r
vectorToMatrix = matrix . (:[]) . unVec

-- De matriz a vector.
matrixToVector :: Matrix r -> Vector r
matrixToVector m | fst (dimension m) == 1 = head (unM m)
                 | otherwise              = error "No pueden tener dimensiones distin

-- Comprobar si la dimensión es correcta según los parámetros que hemos establecido.
(!!!) :: Matrix a -> (Int,Int) -> a
m !!! (r,c) | r >= 0 && r < rows && c >= 0 && c < cols = unMVec m !! r !! c
            | otherwise = error "!!!: Fuera de los límetes"
  where
  (rows,cols) = dimension m
```

Utilizando las funciones anteriores podemos implementar propiedades y operaciones con las matrices.

```
-- | Dimensión de la matriz.
dimension :: Matrix r -> (Int, Int)
dimension (M xs) | null xs   = (0,0)
                 | otherwise = (length xs, length (unVec (head xs)))
```

```
-- | Comprobar si una matriz es cuadrada.
isSquareMatrix :: Matrix r -> Bool
isSquareMatrix (M xs) = all (== length xs) (map lengthVec xs)

-- | Transponer la matriz.
transpose :: Matrix r -> Matrix r
transpose (M xs) = matrix (L.transpose (map unVec xs))

-- | Suma de matrices.
addM :: Ring r => Matrix r -> Matrix r -> Matrix r
addM (M xs) (M ys)
  | dimension (M xs) == dimension (M ys) = m
  | otherwise = error "Las dimensiones de las matrices no pueden ser distintas"
  where
  m = matrix (zipWith (zipWith (<+>)) (map unVec xs) (map unVec ys))

-- | Multiplicación de matrices.
mulM :: Ring r => Matrix r -> Matrix r -> Matrix r
mulM (M xs) (M ys)
  | snd (dimension (M xs)) == fst (dimension (M ys)) = m
  | otherwise = error "Las dimensiones de las matrices no pueden ser distintas"
    where
    m = matrix [ [ mulVec x y | y <- L.transpose (map unVec ys) ]
               | x <- map unVec xs ]

-- | Multiplicación de Vectores
mulVec xs ys | length xs == length ys = foldr (<+>) zero $ zipWith (<**>) xs ys
             | otherwise = error "Las dimensiones de los vectores no pueden ser disti
```

Para utilizar matrices sobre los anillos debemos implementarlo mediante las instancias, pero hay que tener cuidado pues el tamaño de la matriz debe codificarse al dar el tipo, las dimensiones de las matrices tienen que ser las adecuadas para que la suma y multiplicación no den errores, y tenemos que dar el vector neutro para la suma según la dimensión que vayamos a utilizar. Por estos motivos damos el código comentado.

```
{-
instance Ring r => Ring (Matrix r) where
  (<+>) = add
  (<**>) = mul
  neg (Vec xs d) = Vec [ map neg x | x <- xs ] d
  zero  = undefined
-}
```

Introducimos ahora las nociones básicas de la matriz identidad.

```
-- | Construcción de la matriz identidad nxn.
```

```
identity :: IntegralDomain r => Int -> Matrix r
identity n = matrix (xs 0)
  where
  xs x | x == n    = []
       | otherwise = (replicate x zero ++ [one] ++
                      replicate (n-x-1) zero) : xs (x+1)

-- Propiedades de la multiplicación a izquierda y derecha de la matriz identidad.
propLeftIdentity :: (IntegralDomain r, Eq r) => Matrix r -> Bool
propLeftIdentity a = a == identity n `mulM` a
  where n = fst (dimension a)

propRightIdentity :: (IntegralDomain r, Eq r) => Matrix r -> Bool
propRightIdentity a = a == a `mulM` identity m
  where m = snd (dimension a)
```

A continuación vamos a trabajar con matrices sobre anillos conmutativos. Realizaremos suma entre filas y columnas, definiremos el concepto de matriz escalar y dado que estas operaciones no afectan a la dimensión daremos funciones para comprobarlo. Una matriz escalar es una matriz diagonal en la que los elementos de la diagonal principal son iguales.

```
-- | Escalar una fila en la matriz.
scaleMatrix :: CommutRing a => Matrix a -> Int -> a -> Matrix a
scaleMatrix m r s
  | 0 <= r && r < rows = matrix $ take r m' ++ map (s <**>) (m' !! r) : drop (r+1) m'
  | otherwise = error "Escala: índice fuera de los límites"
  where
  (rows,_) = dimension m
  m'       = unMVec m

-- Escalar una matriz no afecta a la dimensión.
propScaleDimension :: (Arbitrary r, CommutRing r) => Matrix r -> Int -> r -> Bool
propScaleDimension m r s = d == dimension (scaleMatrix m (mod r rows) s)
  where d@(rows,_) = dimension m

-- | Intercambiar dos filas de una matriz.
swap :: Matrix a -> Int -> Int -> Matrix a
swap m i j
  | 0 <= i && i <= r && 0 <= j && j <= r = matrix $ swap' m' i j
  | otherwise = error "Intercambio: índice fuera de los límites"
  where
  (r,_) = dimension m
  m'    = unMVec m

  swap' xs 0 0     = xs
  swap' (x:xs) 0 j = (x:xs) !! j : take (j-1) xs ++ x : drop j xs
```

```haskell
  swap' xs i 0     = swap' xs 0 i
  swap' (x:xs) i j = x : swap' xs (i-1) (j-1)

-- Intercambiar filas de una matriz no afecta a la dimensión.
propSwapDimension :: Matrix () -> Int -> Int -> Bool
propSwapDimension m i j = d == dimension (swap m (mod i r) (mod j r))
  where d@(r,_) = dimension m

-- El intercambio es en sí mismo identidad, es decir, intercambiar dos filas ya
-- intercambiadas entre ellas vuelve a estar como al principio.
propSwapIdentity :: Matrix () -> Int -> Int -> Bool
propSwapIdentity m i j = m == swap (swap m i' j') i' j'
  where
  d@(r,_) = dimension m
  i'      = mod i r
  j'      = mod j r


-- Sumar dos filas de la matriz .
addRow :: CommutRing a => Matrix a -> Vector a -> Int -> Matrix a
addRow m row@(Vec xs) x
  | 0 <= x && x < r = matrix $ take x m' ++
                              zipWith (<+>) (m' !! x) xs :
                              drop (x+1) m'
  | c /= length xs  = error "SumaFila: La longitud de la fila es distinta."
  | otherwise       = error "SumaFila: Error al seleccionar la fila."
    where
    (r,c) = dimension m
    m'    = unMVec m

-- Las operaciones de suma entre filas no afectan a la dimensión.
propAddRowDimension :: (CommutRing a, Arbitrary a)
                    => Matrix a -> Vector a -> Int -> Property
propAddRowDimension m row@(Vec xs) r =
  length xs == c ==> d == dimension (addRow m row (mod r r'))
  where d@(r',c) = dimension m

-- Sumar dos columnas de la matriz.
addCol :: CommutRing a => Matrix a -> Vector a -> Int -> Matrix a
addCol m c x = transpose $ addRow (transpose m) c x

-- Restar filas y columnas.
subRow, subCol :: CommutRing a => Matrix a -> Vector a -> Int -> Matrix a
subRow m (Vec xs) x = addRow m (Vec (map neg xs)) x
subCol m (Vec xs) x = addCol m (Vec (map neg xs)) x
```

Gracias a todo lo anterior ahora podemos implementar el método de Gauss-Jordan para poder resolver sistemas $Ax = b$ donde A es una matriz mxn y b es un vector columna de n filas. Comenzaremos por obtener los pivots en cada fila, escalonar la matriz y finalmente hacer el paso de "Jordan"para finalmente conseguir la solución del sistema.

```
-- Multiplicar la fila donde está el pivot y sumarle la fila en la que queremos hacer
pivot :: CommutRing a => Matrix a -> a -> Int -> Int -> Matrix a
pivot m s p t = addRow m (fmap (s <**>) (unM m !! p)) t

-- Encontrar el primer cero en las filas debajo del pivot y devolver el valor y el nú
-- de la fila en la que está.
findPivot :: (CommutRing a, Eq a) => Matrix a -> (Int,Int) -> Maybe (a,Int)
findPivot m (r,c) = safeHead $ filter ((/= zero) . fst) $ drop (r+1) $ zip (head $ dr
  where
  m' = unMVec m

  safeHead []     = Nothing
  safeHead (x:xs) = Just x

fE :: (Field a, Eq a) => Matrix a -> Matrix a
fE (M [])         = M []
fE (M (Vec []:_)) = M []
fE m     = case L.findIndices (/= zero) (map head xs) of
  (i:is) -> case fE (cancelOut m [ (i,map head xs !! i) | i <- is ] (i,map head xs !!
    ys -> matrix (xs !! i : map (zero :) (unMVec ys))
  []     -> case fE (matrix (map tail xs)) of
    ys -> matrix (map (zero:) (unMVec ys))
  where
  cancelOut :: (Field a, Eq a) => Matrix a -> [(Int,a)] -> (Int,a) -> Matrix a
  cancelOut m [] (i,_)    = let xs = unMVec m in matrix $ map tail (L.delete (xs !! i
  cancelOut m ((t,x):xs) (i,p) = cancelOut (pivot m (neg (x </> p)) i t) xs (i,p)

  xs = unMVec m


-- | Calcular la forma escalonada de un sistema Ax = b.
forwardElim :: (Field a, Eq a) => (Matrix a,Vector a) -> (Matrix a,Vector a)
forwardElim (m,v) = fE m' (0,0)
  where
  -- fE coge la matriz a escalonar y la fila y columnas correspondientes.
  fE :: (Field a, Eq a) => Matrix a -> (Int,Int) -> (Matrix a,Vector a)
  fE (M []) _  = error "forwardElim: La matriz dada es vacía."
  fE m rc@(r,c)
      -- El algoritmo se hace cuando llega a la última columna o fila.
    | c == mc || r == mr =
      -- Descompone la matriz en A y b de nuevo.
      (matrix *** Vec) $ unzip $ map (init &&& last) $ unMVec m
```

```
    | m !!! rc == zero    = case findPivot m rc of
      -- Si el pivot de la fila pivot es cero entonces intercambiamos la fila pivot
      -- por la primera fila con elemento no nulo en la columna del pivot.
      Just (_,r') -> fE (swap m r r') rc
      -- Si todos los elementos de la columna pivot son cero entonces nos movemos
      -- a la siguiente columna por la derecha.
      Nothing     -> fE m (r,c+1)

    | m !!! rc /= one     =
      -- Convertir el pivot en 1.
      fE (scaleMatrix m r (inv (m !!! rc))) rc

    | otherwise           = case findPivot m rc of
      -- Hacer 0 el primer elemento distinto de cero en la fila pivot.
      Just (v,r') -> fE (pivot m (neg v) r r') (r,c)
      -- Si todos los elementos en la columna pivot son 0 entonces nos vemos a la
      -- fila de abajo y hacia la columna de la derecha.
      Nothing     -> fE m (r+1,c+1)

  (mr,mc) = dimension m

  -- Combina A y b una matriz donde la última columna es b.
  m' = matrix $ [ r ++ [x] | (r,x) <- zip (unMVec m) (unVec v) ]


-- | Realizar el paso "Jordan" en la eliminación de Gauss-Jordan. Esto es hacer que c
-- elemento encima de la diagonal sea cero.

jordan :: (Field a, Eq a) => (Matrix a, Vector a) -> (Matrix a, Vector a)
jordan (m, Vec ys) = case L.unzip (jordan' (zip (unMVec m) ys) (r-1)) of
  (a,b) -> (matrix a, Vec b)
  where
  (r,_) = dimension m

  jordan' [] _ = []
  jordan' xs c =
    jordan' [ (take c x ++ zero : drop (c+1) x, v <-> x !! c <**> snd (last xs))
            | (x,v) <- init xs ] (c-1) ++ [last xs]


-- | Eliminación por Gauss-Jordan: Dados A y B resuelve Ax=B.
gaussElim :: (Field a, Eq a, Show a) => (Matrix a, Vector a) -> (Matrix a, Vector a)
gaussElim = jordan . forwardElim

gaussElimCorrect :: (Field a, Eq a, Arbitrary a, Show a) => (Matrix a, Vector a) -> P
gaussElimCorrect m@(a,b) = fst (dimension a) == lengthVec b && isSquareMatrix a ==>
  matrixToVector (transpose (a `mulM` transpose (M [snd (gaussElim m)]))) == b
```

4.2. Anillos Coherentes y Fuertemente Discretos

Ya podemos resolver sistemas de ecuaciones con matrices sobre anillos conmutativos, ahora daremos las nociones de anillos coherentes, lo vemos implementado en el módulo `TAHCoherent`

```
module TAHCoherent
  ( Coherent(solve)
  , propCoherent, isSolution
  , solveMxN, propSolveMxN
  , solveWithIntersection
  , solveGeneralEquation, propSolveGeneralEquation
  , solveGeneral, propSolveGeneral
  ) where

import Test.QuickCheck

import TAHIntegralDomain
import TAHStronglyDiscrete
import TAHMatriz
import TAHIdeal
```

Definición 10. *Un anillo R es coherente si todo ideal generado finitamente es finito. Esto significa que dado una matriz $M \in R^{1\times n}$ existe una matriz $L \in R^{n\times m}$ para $m \in \mathbb{N}$ tal que $ML = 0$ y*

$$MX = 0 \Leftrightarrow \exists Y \in R^{m\times 1}.\ X = LY$$

De esta forma es posible calcular un conjunto de generadores para soluciones de ecuaciones en un anillo coherente. En otras palabras, el conjunto de soluciones para $MX = 0$ esta generado finitamente. Comenzamos por establecer la clase de los anillos coherentes:

```
class IntegralDomain a => Coherent a where
  solve :: Vector a -> Matrix a
```

Empezamos por dar funciones para comprobar soluciones y resolver sistemas de ecuaciones sobre anillos conmutativos.

```
-- | Test para comprobar que la segunda matriz es una solución de la primero.
isSolution :: (CommutRing a, Eq a) => Matrix a -> Matrix a -> Bool
isSolution m sol = all (==zero) (concat (unMVec (m `mulM` sol)))
```

```
propCoherent :: (Coherent a, Eq a) => Vector a -> Bool
propCoherent m = isSolution (vectorToMatrix m) (solve m)

-- | Resolver un sistema de ecuaciones.
solveMxN :: (Coherent a, Eq a) => Matrix a -> Matrix a
solveMxN (M (l:ls)) = solveMxN' (solve l) ls
  where
  -- Resolver de forma recursiva todos los subsistemas. Si la solución calculada es d
  -- hecho una solución del siguiente conjunto de ecuaciones, entonces no hagas nada.
  -- Esto resuelve los problemas que tienen muchas filas idénticas en el sistema,
  -- como [[1,1], [1,1]].
  solveMxN' :: (Coherent a, Eq a) => Matrix a -> [Vector a] -> Matrix a
  solveMxN' m []      = m
  solveMxN' m1 (x:xs) = if isSolution (vectorToMatrix x) m1
                         then solveMxN' m1 xs
                         else solveMxN' (m1 `mulM` m2) xs
    where m2 = solve (matrixToVector (mulM (vectorToMatrix x) m1))


-- | Test para comprobar que la solución de un sistema MxN es de hecho una solución d
propSolveMxN :: (Coherent a, Eq a) => Matrix a -> Bool
propSolveMxN m = isSolution m (solveMxN m)
```

Implementamos la solución por intersección. Vamos a ver que si hay un algoritmo para calcular un f.g. conjunto de generadores para la intersección de dos f.g. ideales entonces el anillo es coherente. Cogemos el vector a resolver, $x_1, .., x_n, yunafuncin$(int)$quecalculelainters$

```
solveWithIntersection :: (IntegralDomain a, Eq a)
                      => Vector a
                      -> (Ideal a -> Ideal a -> (Ideal a,[[a]],[[a]]))
                      -> Matrix a
solveWithIntersection (Vec xs) int = transpose $ matrix $ solveInt xs
  where
  solveInt []     = error "solveInt: No se puede resolver un sistema vacío"
  solveInt [x]    = [[zero]] -- Caso base, podría ser [x, y] también ...
                             -- Eso no daría la solución trivial ...
  solveInt [x,y]  | x == zero || y == zero = [[zero,zero]]
                  | otherwise =
    let (Id ts,us,vs) = (Id [x]) `int` (Id [neg y])
    in [ u ++ v | (u,v) <- zip us vs ]
  solveInt (x:xs)
    | x == zero             = (one : replicate (length xs) zero) : (map (zero:) $ sol
    | isSameIdeal int as bs = s ++ m'
    | otherwise             = error "solveInt: No puede calcularse la intersección"
      where
      as            = Id [x]
```

```
    bs              = Id (map neg xs)

    -- Calculamos la intersección de <x1> and <-x2,...,-xn>
    (Id ts,us,vs) = as ‘int‘ bs
    s               = [ u ++ v | (u,v) <- zip us vs ]

    -- Resuelve <0,x2,...,xn> recursivamente
    m               = solveInt xs
    m’              = map (zero:) m
```

La propiedad de ser fuertemente discreto es una propiedad muy fuerte que pueden poseer los anillos. Pues si el anillo es muy discreto y coherente, no solo es posible resolver sistemas como $MX = 0$, también es posible resolver sistemas generales del tipo $MX = A$.

Si R es un dominio de integridad coherente fuertemente discreto entonces es posible resolver sistemas lineales arbitrarios. Dado $MX = A$ es posible calcular X_0 y L tal que $ML = 0$ y

$$MX = A \leftrightarrow \exists\ YX = LY + X_0$$

Implementamos la resolución de estos anillos coherentes fuertemente discretos:

```
-- | Anillos coherentes fuertemente discretos.
solveGeneralEquation :: (Coherent a, StronglyDiscrete a) => Vector a -> a -> Maybe (M
solveGeneralEquation v@(Vec xs) b =
  let sol = solve v
  in case b ‘member‘ (Id xs) of
    Just as -> Just $ transpose (M (replicate (length (head (unMVec sol))) (Vec as)))
                      ‘addM‘ sol
    Nothing -> Nothing


propSolveGeneralEquation :: (Coherent a, StronglyDiscrete a, Eq a)
                         => Vector a
                         -> a
                         -> Bool
propSolveGeneralEquation v b = case solveGeneralEquation v b of
  Just sol -> all (==b) $ concat $ unMVec $ vectorToMatrix v ‘mulM‘ sol
  Nothing  -> True


isSolutionB v sol b = all (==b) $ concat $ unMVec $ vectorToMatrix v ‘mulM‘ sol

-- | Resolver un sistema general de ecuaciones lineales de la forma AX=B.
-- A es una matriz dada y B viene dada como un vector fila
-- (este debería ser un vector columna).
```

```
solveGeneral :: (Coherent a, StronglyDiscrete a, Eq a)
             => Matrix a   -- M
             -> Vector a   -- B
             -> Maybe (Matrix a, Matrix a)  -- (L,X0)
solveGeneral (M (l:ls)) (Vec (a:as)) =
  case solveGeneral' (solveGeneralEquation l a) ls as [(l,a)] of
    Just x0 -> Just (solveMxN (M (l:ls)), x0)
    Nothing -> Nothing
  where
  -- Calculamos una nueva solución de forma inductiva y verificando que la nueva solu
  -- cumple todas las ecuaciones anteriores.
  solveGeneral' Nothing _ _ _                = Nothing
  solveGeneral' (Just m) [] [] old           = Just m
  solveGeneral' (Just m) (l:ls) (a:as) old =
    if isSolutionB l m a
       then solveGeneral' (Just m) ls as old
       else case solveGeneralEquation (matrixToVector (vectorToMatrix l `mulM` m)) a
         Just m' -> let m'' = m `mulM` m'
                    in if all (\(x,y) -> isSolutionB x m'' y) old
                          then solveGeneral' (Just m'') ls as ((l,a):old)
                          else Nothing
         Nothing -> Nothing
  solveGeneral' _ _ _ _ = error "solveGeneral: La entrada no es válida"


-- Estaría bien poder generar solo sistemas con solución ...
propSolveGeneral :: (Coherent a, StronglyDiscrete a, Eq a) => Matrix a -> Vector a ->
propSolveGeneral m b = length (unM m) == length (unVec b) ==> case solveGeneral m b o
  Just (l,x) -> all (==b) (unM (transpose (m `mulM` x))) &&
                isSolution m l
  Nothing -> True
```

Bibliografía

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

Índice alfabético