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. 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 mediante programación funcional y teoría de tipos.

3.1. Anillos

Comenzamos dando las primeras definiciones y propiedades básicas sobre anillos 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 cuál comprende las operaciones con listas, y *Test.QuickCheck* que contine las funciones para comprobar 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 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, $(neg\ x)$ representará el inverso de x respecto de la suma $< + >$. Todas ellas se concretarán para cada anillo particular.

Mediante los operadores $infixl$ e $infixr$ se puede establecer el orden de precedencia (de 0 a 9) de una operación, así como la asociatividad de dicha operación ($infixl$ para asociatividad por la izquierda, $infixr$ para asociatividad por la derecha e $infix$ para no asociatividad). En este caso, las declaraciones ($infixl\ 6\ < + >$ e $infixl\ 7\ < ** >$) hacen referencia a la asociatividad por la izquierda de ambas operaciones, siendo 6 el nivel de precedencia de $< + >$ y 7 el nivel de precedencia de $< ** >$.

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

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

Una vez declarado el tipo y la signatura de las funciones, 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 objetos se ajusta la expresión.

Todos los axiomas que tenemos que introducir tienen la misma estructura, reciben un tipo de la clase *Ring* y *Eq* para devolver elementos del tipo *Bool* y *String*.

La clase *Ring* la acabamos de definir y la clase *Eq* es la clase de los tipos con 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*].

```
-- |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 definimos 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, 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:
--type Zint = Integer

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, suma, producto y potencia. 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 hemos definimos la suma 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. 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*.

```
-- |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. Visto desde el punto de vista de la programación funcio-

nal,un anillo conmutativo es una subclase de la clase *Ring*. Solo necesitaremos una función para definirlo. Damos primero su definición.

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 (Show a, 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 definimos una propiedad que compruebe, en cada caso particular, que las operaciones concretas de una instancia verifiquen 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
```

Un ejemplo de anillo conmutativo es el de los números enteros, el cual definimos sus operaciones en el anterior módulo de anillos. Por tanto añadiendo la instancia a la clase de anillos conmutativos, comprueba que se verifiquen las operaciones necesarias para ser un anillo conmutativo.

```
instance CommutRing Integer
```

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 vamos a definir los dominios de integridad sobre 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 la especificación 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 también 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. 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 para ideales arbitrarios no es adecuada para definirla de forma constructiva. En la implementación en Haskell, nos reduciremos a los ideales finitamente generados.

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 $E =< e_1, .., e_r >:= \{a_1e_1 + .. + a_re_r \mid a_1, ..a_r \in A\}$. 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 $x = \alpha_1x_1 + .. + \alpha_rx_r$.

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. Mediante *data* se define el tipo *Ideal a*, donde *a* es un tipo cualquiera que representa los elementos del ideal. El constructor es *Id* cuyo conjunto es una lista de elementos de *a* (los generados del ideal).

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 $(Id\ [one])$.

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 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]
```

Al añadir $deriving(Show)$ 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 $< 2,5 >$.

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 nos ayudará a comprobar las funciones a verificar para ser un ideal.

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*).

Finalmente hemos implementado uno de los ideales canónicos, el ideal cero, $< 0 >$. A continuación, damos la definición 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 $\mathbb{Z}$ en los cuales todos los ideales son principales se llaman clásicamente dominios de ideales principales. Pero constructivamente esta definición no es adecuada. Sin embargo, nosotros solo queremos considerar anillos en los cuales todos los ideales finitamente generados son principales. Al ser representados por un conjunto finito, podemos implementarlo a nivel computacional. Estos anillos se llaman dominios de Bézout y se considerarán en el siguiente capítulo. Siempre que se pueda añadiremos ejemplos sobre los enteros, haciendo uso de la instancia sobre los enteros especificada en los anteriores módulos.

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

--Ejemplos:
```

```
--> isPrincipal (Id [2,3])
--False
--> isPrincipal (Id [4])
--True
```

Mediante la función $from\ Id$, definida a continuación, mostramos la lista de los generadores de $(Id\ xs)$.

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

--Ejemplos:
--> fromId (Id [3,4])
--[3,4]
--> fromId (Id [4,5,8,9,2])
--[4,5,8,9,2]
```

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 $I + J = \{a + b \mid a \in I, b \in J\}$.La suma de ideales también es un ideal.*

Está definición es para cualquier ideal, nosotros nos centramos en los ideales finitamente generados. La suma de ideales finitamente generados es también un ideal finitamente generado y esta puede obtenerse a partir de los generadores de ambos ideales, es decir, $I + J = < I \cup J >$.

```
addId :: (CommutRing a, Eq a) => Ideal a -> Ideal a -> Ideal a
addId (Id xs) (Id ys) = Id (nub (xs ++ ys))

--Ejemplos:
--> addId (Id [2,3]) (Id [4,5])
-- <2,3,4,5>
--> addId (Id [2,3,4]) (Id [3,4,6,7])
-- <2,3,4,6,7>
```

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

De igual forma que en la suma, está es la definición general para cualquier ideal. Centrándonos en los ideales finitamente generados, el producto de ideales finitamente generados es también un ideal finitamente generado y este se obtiene de igual forma

que para cualquier ideal, solo que el producto es entre ideales finitamente generados.

```
-- | 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 ]

--Ejemplos:
--> mulId (Id [2,3]) (Id [4,5])
-- <8,10,12,15>
--> mulId (Id [2,3,4]) (Id [3,4,6,7])
-- <6,8,12,14,9,18,21,16,24,28>
```

A continuación veremos una función cuyo objetivo 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 $(Id\, zs)$, z_k tiene una expresión como combinación lineal de *xs* e *ys*, cuyos coeficientes vienen dados por *as* y *bs*, 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 detalle 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 $(Id\, zs, as, bs)$, 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. Cada elemento de *zs* se puede expresar como combinación lineal de *xs* con

los coeficientes proporcionados por el correspondiente elemento de *as* (análogamente, como combinación lineal de *ys* con los coeficientes proporcionados por *bs*).

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])

--Ejemplo:
--> zeroIdealWitnesses [2,3] [4,5]
--(<0>,[[0,0]],[[0,0]])
```

Capítulo 4

Vectores y Matrices

En este capítulo implementaremos los vectores y matrices como listas para poder construir los módulos siguientes. En Haskell existen ya estas librerías pero solo podemos usarlas si nos restringimos al tipo numérico. Por ello, crearemos este módulo con el fin de generalizar los conceptos para cualquier tipo de anillo.

4.1. Vectores

Antes de introducir las matrices tenemos que especificar ciertas operaciones y funciones sobre los vectores, pues las filas de las matrices serán vectores, así tendremos la matriz como lista de listas. Damos los conceptos necesarios en el en el módulo `TAHVector`Para comenzar este módulo, necesitaremos hacer uso de librerías de Haskell como *Control.Monad* para utilizar ciertas funciones que nos ayuden a construir nuestros vectores.

```
module TAHVector
  ( Vector(Vec)
  , unVec, lengthVec
  , sumVec, mulVec
  ) where

import Control.Monad (liftM)

import Test.QuickCheck
import TAHField
```

Comenzamos por implementar las nocines básicas de los vectores. Creamos un nuevo tipo para definir un vector, usaremos las listas para trabajr con los vectores en Has-

kell. Daremos también una función para generar vectores de forma aleatoria, para poder comprobar resultados mediante *QuickCheck*.

```
-- | 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)
```

Explicamos brevemente algunas de las funciones utilizadas en el generador de vectores, con *choose* se elige de forma aleatoria un número, en nuestro caso, entre 1 y 10. La función $liftM$ nos permite transformar una función en una función correspondiente dentro de otra configuración en nuestro caso en forma de vector y junto con *gen* que es una función que genera un tipo dado de forma aleatoria en nuestro caso un vector gracias a $liftM$.

Para trabajar con vectores, haremos uso de la clase de tipos *Functor* de Haskell, importada mediante $(import\ Data.Function\ (on))$. Esta clase de tipos está implementada de la siguiente forma:

```
-- class Functor f where
--     fmap :: (a -> b) -> f a -> f b
```

Define una función, $fmap$, y no proporciona ninguna implementación por defecto. El tipo de $fmap$ es similar al tipo de *map*. Aquí, f no es un tipo concreto, sino un constructor de tipos que toma un tipo como parámetro. Vemos que $fmap$ toma una función de un tipo a otro y un funtor aplicado a un tipo y devuelve otro funtor aplicado con el otro tipo.

En nuestro caso crearemos una instancia con la clase de tipos *Functor* sobre el cons-

tructor *Vector* con el objetivo de establecer los vectores en forma de lista. Definiremos la función $(fmap\ f)$ como tipo vector mediante *Vec* y mediante aplicaremos *unVec* (una función que definiremos más adelante) a la lista obtenida tras aplicar la función $(map\ f)$.

```
instance Functor Vector where
  fmap f = Vec . map f . unVec
```

Veamos las operaciones de suma y multiplicación de vectores sobre anillos, para ello restringimos a los anillos conmutativos de forma general. La multiplicación de vectores es el producto escalar de ambos. Añadimos unos ejemplos sobre los números enteros.

```
sumVec:: Ring a => Vector a -> Vector a -> Vector a
sumVec (Vec xs) (Vec ys) | length xs == length ys = Vec (zipWith (<+>) xs ys)
             | otherwise = error "Los vectores tienen dimensiones diferentes"

mulVec:: Ring a => Vector a -> Vector a -> a
mulVec (Vec xs) (Vec ys) | length xs == length ys = foldr (<+>) zero $
                           zipWith (<**>) xs ys
             | otherwise = error "Los vectores tienen dimensiones diferentes"

-- | Ejemplos:
--> sumVec (Vec [2,3]) (Vec [4,5])
--   [6,8]
--> mulVec (Vec [2,3]) (Vec [4,5])
--   23
```

Para definir la suma y multiplicación de vectores sobre un anillo en concreto, estas operaciones no son sufientes, pues hay que añadir el resto de axiomas del anillo. Así como el elemento neutro para suma y producto, y la función inversa *neg*. Tenemos que dejarlas comentadas pues para que funcionen tenemos que asignar el elemento neutro para la suma y producto, pues estamos definiéndolo 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) = sumVec (Vec xs) (Vec ys)
  (Vec xs) <**> (Vec ys) = mulVec (Vec xs) (Vec ys)
  one  = ?
  zero = ?
  neg = ?
-}
```

Para finalizar, damos la función que muestra el vector en forma de lista y la que mide la longitud de un vector en ese formato. Acompañamos de ejemplos para mostrar los resultados que se obtienen.

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

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

-- | Ejemplos:
--> unVec (Vec [2,3])
--   [2,3]
--> lengthVec (Vec [3,4,5])
--   3
```

4.2. Matrices

Una vez dadas las funciones y operaciones de los vectores podemos comenzar a implementar las matrices, notesé que cada fila o columna de una matriz puede verse como un vector. Nuestra idea es dar las matrices como una lista de listas de forma que las listas sean las filas de la matriz que serán vectores. El objetivo de esta sección es implementar el método de Gauss-Jordan con el fin de poder resolver sistemas de la forma $Ax = b$. Lo vemos en el módulo `TAHMatrix`Antes de comenzar, importaremos algunas librerías necesarias para construir las matrices. Entre ellas utilizaremos $'Control.Arrow'$, de ella tenemos que excluir la suma $(< + >)$ pues nosotros utilizamos la definida en anillos.

```
module TAHMatrix
  ( 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 Control.Monad (liftM)
import Control.Arrow hiding ((<+>))
```

```
import Test.QuickCheck

import TAHField
import TAHVector
```

Declaramos el nuevo tipo de matrices, la representación de la matriz viene dada en forma de lista de vectores, donde cada vector de la lista representa una fila de la matriz. Daremos también una instancia para que se puedan mostrar. Así como un generador de matrices aleatorio, similar al utilizado en vectores, con el fin de poder comprobar resultados mediante *QuickCheck*.

```
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 a lo sumo 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)
```

Del mismo modo que para vectores, para matrices volveremos a utilizar la clase de tipos *Functor* para establecer matrices en forma de listas.

```
instance Functor Matrix where
  fmap f = M . map (fmap f) . unM
```

Una vez implementado el tipo de las matrices vamos a definir la función para construir una matriz de dimensión *mxn* a partir de una lista de listas, de forma que cada lista es una fila de la matriz (todas de la misma longitud) y la longitud de un lista es el número de columnas. Vamos a dar una función para que devuelva la matriz de la forma en que visualmente la vemos, es decir una fila debajo de la otra.

```
-- | 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 obtener una submatriz.

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

-- | Aplicamos la función 'unM' 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 distintas"

-- | Obtener una submatriz.
(!!!) :: 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. Daremos la dimensión de la matriz, una función que compruebe si la matriz es cuadrada, es decir, que el número de filas coincide con el número de columnas. Y la función para transponer una matriz, es decir, pasar las filas a columnas. Para esta última, utilizaremos la función *'transpose'* de la librería *Data.List* que se aplica sobre listas de forma que agrupa los elementos para que las listas de filas estén en la posición de la columna correspondiente.

```
-- | 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 (== l) (map lengthVec xs)
                      where l = length xs

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

La suma y multiplicación la restringiremos a la clase de los anillos, *Ring*. La suma de matrices recordamos que da una matriz cuyas entradas son la suma de las entradas correspondientes de las matrices a sumar. Para esta suma lo haremos mediante suma de listas ya que utilizaremos la función *'matrix'* para mostrar la matriz correspondiente y esta recibe como argumento de entrada una lista de listas.

Por otro lado la multiplicación de matrices recordamos que consiste en multiplicar cada fila de la primera matriz por cada columna de la segunda matriz, obteniendo así un elemento en la entrada correspondiente al par de la fila y columna multiplicada. Aquí si podemos utilizar la operación *'mulVec'* entre vectores pues devuelve un valor no un vector, por tanto obtenemos una lista de listas. Recordamos que para poder realizar multiplicaciones entre matrices el número de columnas de la primera matriz tiene que ser igual al número de filas de la segunda matriz.

```
-- | 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 son 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 "La dimensión de colunmas y filas es distinta"
    where
    m = matrix [ [mulVec x y | y <- unM (transpose (M ys)) ]
               | x <- unM (M xs) ]
```

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
  one = undefined
-}
```

Introducimos las propiedades básicas de la matriz identidad. Estas estarán restringidas a la clase de dominio de integridad, *IntegralDomain*.

```
-- | 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, por ello restringiremos a la clase *CommutRing*. Realizaremos operaciones entre filas y columnas, y veremos que estas operaciones no afectan a la dimensión de la matriz. Hacemos esta restricción debido a que uno de los objetivos es poder aplicar el método de Gauss-Jordan, y para ello exigiremos que las matrices pertenezcan a anillos conmutativos, debido a que en el siguiente capítulo necesitaremos estas operaciones con matrices y estamos restringidos a anillos conmutativos.

Damos una breve descripción de la primera operación. El objetivo es multiplicar una fila por un escalar, de forma que la matriz obtenida sea la misma con la fila que queramos modificada como el resultado de multiplicar esta fila por un número o escalar, quedando el resto de filas sin modificar, los argumentos de entrada serán la matriz, el número de la fila que queremos modificar menos 1 (pues la función $(!!r)$ de la librería Data.List selecciona el elemento $r+1$ de la lista, pues es un índice y comienza en 0). Comprobaremos que esta operación no afecta a la dimensión, pues la dimensión de la matriz resultante es la misma que la primera matriz.

```
-- | Multiplicar una fila por un escalar 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 "La fila escogida es mayor que la dimensión"
  where
  (rows,_) = dimension m
  m'       = unMVec m

-- La dimensión no varía.
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
```

La siguiente operación consiste en intercambiar filas, el objetivo de $(swap\ m\ i\ j)$ es dada una matriz m intercambiar las filas i y j, de forma que la fila i acabe en la posición de la fila j y viceversa. Comprobamos que no varía de dimensión. Comprobaremos también que si realizamos el mismo intercambio dos veces obtenemos la matriz inicial.

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

-- Al intercambiar filas de una matriz no varía 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.
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
```

Mediante la función $(addRow\ m\ row@(Vecxs)\ x)$ realizamos la operación de sumar un vector $(row@(Vecxs))$ a la fila $x+1$ de una matriz m dada, recordamos que es importante que el vector tenga la misma dimensión que el número de colunmas de la matriz. Verificamos que la dimensión no varía. Seguidamente realizamos la misma operación sobre las columnas, para ello basta transponer la matriz y aplicar la función sobre las filas.

```
-- | Sumar un vector a una fila 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 /= lengthVec row  = 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

-- La operación 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 un vector columna a una columna de la matriz.
addCol :: CommutRing a => Matrix a -> Vector a -> Int -> Matrix a
addCol m c x = transpose $ addRow (transpose m) c x
```

Finalmente, realizaremos la operación anterior pero esta vez restando un vector, es decir, la matriz resultante mantiene todas las filas menos la fila seleccionada, a la cuál se le resta un vector dado.

```
-- Restar un vector fila o columna a una fila o colunma, respectivamente,
-- de una matriz.
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. Para ello exigiremos que las matrices pertenezcan a anillos conmutativos. Comenzaremos por obtener los pivotes en cada fila, escalonar la matriz y finalmente hacer el paso de Jordan para finalmente conseguir la solución del sistema. El paso de Jordan consiste en hacer ceros por encima de la diagonal de la matriz cuando previamente se ha obtenido ceros debajo de la diagonal de la matriz, esta primera parte se conoce como aplicar Gauss o realizar el método de Gauss.

Para empezar damos las funciones de obtener un 0 en la posición del pivote y la segunda función consiste en localizar el siguiente pivote.

```
-- | Multiplicar por el escalar correspondiente  la fila donde está
-- el pivote y sumarle la fila en la que queremos hacer un 0.
-- El escalar se elige de forma que al sumar la fila consigamos un 0
-- en la posición del pivote.
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 número 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 $ drop c $ unMVec $ transpose m) [0..]
  where
  m' = unMVec m

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

Con la siguiente función buscamos modificar la fila del pivote de forma que obtengamos el cero en la posición del pivote. Para ello necesitamos que sea un cuerpo pues necesitamos que exista inverso, ya que el valor del escalar al multiplicarse por la fila en la posición del pivote debemos obtener el inverso del pivote para que al sumarlo obtengamos un cero.

```
-- | Modificamos las filas para hacer 0 en la columna del pivot.
```

```
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 !! i)) of
    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) xs)
  cancelOut m ((t,x):xs) (i,p) =
                 cancelOut (pivot m (neg (x </> p)) i t) xs (i,p)

  xs = unMVec m
```

Para calcular la forma escalonada seguimos necesitando que los elementos de las matrices pertenezca a un cuerpo. Primero aplicamos Gauss, es decir, obtenemos ceros por debajo de la diagonal.

```
-- | 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 toma como argumento de entrada la matriz a escalonar y
  --la posición del pivote.
  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

  -- Divide la matriz en A y b, donde b es la última columna de la matriz
  -- y A el resto de la matriz.
  m' = matrix $ [ r ++ [x] | (r,x) <- zip (unMVec m) (unVec v) ]
```

En segundo lugar aplicamos el paso de Jordan que consiste en obtener ceros por encima de la diagonal. Para aplicar el método de Gauss-Jordan es necesario aplicar antes el paso de Gauss y después el de Jordan.

```
-- | 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]
```

Finalmente, podemos realizar el método de Gauss-Jordan, con el objetivo de resolver un sistema de ecuaciones de la forma $Ax = b$, primero obtenemos la matriz con solo la diagonal que es la obtenida tras aplicar Gauss y luego Jordan esto lo obtenemos con la función de denotaremos *gaussElim*.

Con la función que denotaremos *gaussElimCorrect* recibe la partición de la matriz A y la matriz b con ella comprobamos que las dimensiones de ambas son correctas de esta forma hemos obtenido la solución del sistema, ya que en la diagonal nos ha quedado nuestro vector de incógnitas con coeficiente igual a 1 y en b se encuentra los valores de nuestras incógnitas.

```
-- | 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
```

Capítulo 5

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. Antes de comenzar introduciremos la noción de anillo fuertemente discreto.

5.1. Anillos Fuertemente Discretos

En esta breve sección mostraremos la noción de anillo discreto y fuertemente discreto, lo vemos en el módulo `TAHStronglyDiscrete`

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

import TAHCommutative
import TAHIdeal
```

Para desarrollar 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 que consideremos serán discretos. Pero hay muchos ejemplos de anillos que no son discretos. Por ejemplo, $\mathbb{R}$ 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.*

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]
```

Hemos creado la función *member* con la cual, mediante el constructor *Maybe* podemos 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 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 particular, si pertenece, devuelve $(Just\ as)$ donde as es la lista de coeficientes de la expresión de x como combinación lineal de los generadores del ideal y, en caso contrario, *Nothing*. Si esto se verifica devuelve *True* y nuestro anillo es fuertemente discreto.

5.2. Anillos Coherentes

En esta sección, nos ayudaremos del módulo de vectores y matrices creado en el capítulo anterior para construir, en Haskell, la noción de anillo coherente con el objetivo de poder resolver sistemas de ecuaciones. Lo vemos en el módulo `TAHCoherent`

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

```
import Test.QuickCheck

import TAHIntegralDomain
import TAHIdeal
import TAHStronglyDiscrete
import TAHMatrix
```

Definición 12. *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 $L \in \mathbb{R}^{n\times m}$ 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
```

Damos funciones para comprobar soluciones y resolver sistemas de ecuaciones sobre anillos conmutativos, para ello usaremos recursión.

```
-- | Test para comprobar que la segunda matriz es una solución de la primera.
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)
```

Con la siguiente función podemos resolver de forma recursiva todos los subsistemas para obtener mediante recursión la solución X. Si la solución calculada es de hecho una solución del siguiente conjunto de ecuaciones, entonces no hace nada. Gracias a esto resolvemos los problemas que tienen muchas filas idénticas en el sistema, como [[1,1], [1,1]].

```
solveMxN :: (Coherent a, Eq a) => Matrix a -> Matrix a
solveMxN (M (l:ls)) = solveMxN' (solve l) ls
  where
  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 obtenido con $solveMxN$ es
propSolveMxN :: (Coherent a, Eq a) => Matrix a -> Bool
propSolveMxN m = isSolution m (solveMxN m)
```

Vemos como funciona $solveMxN$ con más detalle. Toma una matriz de entrada y selecciona el primer vector (o la primera fila de la matriz) para aplicar $solveMxN'$. Esta toma dos matrices de entrada, veamos el caso de $(solveMxN'\ m1\ (x : xs))$, comprobamos que la matriz $m1$ es solución de x. En esta caso, resolvemos con $(solveMxN' m1 xs)$ aplicando la recursión. En caso contrario, multiplicamos la matriz $m1$ por la matriz x, la matriz resultante la transformamos en vector y aplicamos $solve$ sobre ella. A continuación se aplica $solveMxN'$ sobre la matriz resultante tras aplicar solve al vector obtenido y sobre xs.

Si hay un algoritmo para calcular un conjunto de generadores finitamente generados para la intersección de dos ideales finitamente generados entonces el anillo es coherente.

Prueba: Cogemos el vector a resolver, $[x1, ..., xn]$, y una función (int) que calcule la intersección de dos ideales.
Si $[x_1, ..., x_n]$ 'int' $[y_1, ..., y_m] = [z_1, ..., z_l]$.
Entonces (int) debería devolver us y vs tales que:
$z_k = n_k1 * x_1 + ... + u_kn * x_n = u_k1 * y_1 + ... + n_km * y_m$

Así podemos obtener una solución del sistema mediante la intersección.

```
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 puede resolver un sistema vacío"
  solveInt [x]    = [[zero]] -- Caso base, podría ser [x,y] también...
                             -- Este 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 se puede calcular la intersección"
    where
    as            = Id [x]
    bs            = Id (map neg xs)

    -- Calculamos al intersección de <x1> y <-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
```

De este código destacamos la función *solveInt*, vamos a explicar brevemente como funciona. Es una función recursiva, obviando los casos base, dada una lista $(x : xs)$ hay 3 casos por orden de prioridad. En primer lugar si x es 0 ejecuta las órdenes de añadir la lista que comienza por 1 seguida de tantos ceros como longitud de la lista xs y esta pequeña lista se añade al comienzo de la segunda lista formada por añadir un 0 a la solución obtenida mediante recursión de la lista xs con $(solveInt\ xs)$. En segundo lugar, en el caso de que x sea distinto de 0, aplicamos *isSameIdeal* para comprobar que la intersección se pueda hacer si resulta ser *True* aplicamos la concatenación de la lista s con m'. Donde s se obtiene a partir de la intersección del ideal $< x >$ y del ideal obtenido tras aplicar *neg* a cada elemento de xs y m' es la lista tras añadir 0 a cada lista de la matriz m.

Dentro de los anillos anillos coherentes, encontramos los anillos coherentes fuertemente discretos. Restringiendo la clase de anillos a la clase de fuertemente discreto, esto nos permite una facilidad mayor a la hora de resolver sistemas. Puesto que, si un anillo es fuertemente discreto y coherente entonces podemos resolver cualquier ecuación del tipo $AX = b$.

```
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
```

La primera función, *solveGeneralEquation* calcula una solución general. Para ello toma un vector (*Vecxs*) el cuál se llamará *v* gracias al @ y el vector solución *b*. A continuación llamamos *sol* a *solvev*, de esta forma pasamos de vector a matriz gracias a *solve*. Seguidamente comprobamos si *b* pertenece al ideal de *xs* en ese caso guardamos en as

–explicar el just y nothing—

Con la siguiente función comprobaremos si la solución obtenida anteriormente (*sol*) es correcta. Para ello comprobaremos que al realizar la multiplicación de nuestra matriz A por la solución obtenida coincide componente a componente con el vector solución *b*.

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

Ahora vamos a resolver sistemas lineales generales de la forma $AX = B$.

```
solveGeneral :: (Coherent a, StronglyDiscrete a, Eq a)
             => Matrix a   -- A
             -> 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 verificamos que la nueva solu
  -- satisface 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: Error en la entrada"
```

De *solveGeneral* explicaremos con detalle como funciona *solveGeneral′* —Explicar hay mucho Nothing y Just—

Con la siguiente propiedad, comprobaremos que la solución es correcta. Primero tenemos que comprobar que las filas de A (se presentará en código por m) son de la misma longitud que b. Después, multiplicamos la matriz A con la matriz solución X y vemos si coincide componente a componente con el vector solución b. Devolviendo al final un *True*.

```
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