Álgebra constructiva en Haskell

Facultad de Matemáticas
Departamento de Ciencias de la Computación e Inteligencia Artificial
Trabajo Fin de Grado

Ángela González Martín

Agradecimientos

A mis padres quiero agradecerles la ayuda y apoyo que me han dado durante esta bonita etapa. Siempre han estado para mí cuando más lo he necesitado, gracias por haberme facilitado el poder estudiar Matemáticas, por ayudarme en los momentos difíciles y por apoyarme en mis decisiones.

A mi tutora María José, por toda la paciencia que ha tenido conmigo y la gran ayuda que me ha brindado para poder realizar este trabajo.

Por último y no menos importante, a todos mis amigos y todas las personas que son importantes para mí que han recorrido este camino conmigo. Ellos han hecho que haya sido una de las mejores experiencias.

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

María José Hidalgo Doblado

Abstract

The objective of this proyect is to represent the algebraic structures in a functional programming language. For this, we have maked use of Haskell. Through the use of type classes it has been specified the notion of ring, commutative ring, integral domain, field, ideal, strongly discrete ring, coherent ring and finally strongly discrete and coherent ring. In the last one it is possible to solve systems of equations, for which it has developed a library of vectors and matrices which elements are in a generic structure (ring, integral domain or field). Furthermore, through an instantiation process it can get specificed examples of the structures represented.

Resumen

El objetivo de este trabajo es representar las estructuras algebraicas en un lenguaje de programación funcional. Para ello, hemos usado Haskell. Mediante el uso de clases de tipos se han especificado las nociones de anillo, anillo conmutativo, dominio de integridad, cuerpo, ideal, anillo fuertemente discreto, anillo coherente y finalmente anillo fuertemente discreto y coherente. En este último es posible resolver sistemas de ecuaciones, para lo cual se ha desarrollado una librería de vectores y matrices cuyos elementos pertenecen a una estructura genérica (anillo, dominio de integridad o cuerpo). Además, mediante un proceso de instanciación se pueden obtener ejemplos concretos de las estructuras representadas.

Índice general

Introducción

Este trabajo tiene como objetivo implementar conceptos de álgebra constructiva en Haskell, desde la noción de anillo hasta la de anillo coherente. El camino a seguir consiste en desarrollar los siguientes puntos:

- Familiarización con las funciones predefinidas en Haskell. Aquí aportaremos las funciones básicas de Haskell con el fin de facilitar la lectura del código, pues muchas de ellas son utilizadas a lo largo del trabajo. Algunas de ellas las podemos encontrar en la librería `Data.List` de Haskell.

- Implementación de la teoría de anillos en Haskell. Introduciendo la noción de anillo, para posteriormente incluir los anillos conmutativos, dominio de integridad, cuerpos e ideales. Hemos definido los conceptos mediante programación funcional ([1] y [7]) y teoría de tipos ([2] y [9]). De tal forma, que mediante las instancias podemos trabajar con cualquier anillo en concreto, como el anillo de los enteros, Z. Para la elaboración de este capítulo se ha utilizado el material de la asignatura "Estructuras Algebraicas" del Departamento de Álgebra de la facultad de Matemáticas, "Tema 3: Anillos" ([5]). También se ha utilizado la tesis "Constructive Algebra in Functional Programming and Type Theory" de Anders Mörtberg ([6]).

- Librería de vectores y matrices sobre anillos en Haskell. Tomando como referencia la actual librería de matrices de la que Haskell dispone, `Data.Matrix`, junto con la tesis de Anders Mörtberg ([6]). En este capítulo, comenzamos con las operaciones básicas entre matrices hasta poder aplicar el Método de Gauss, trabajando con vectores en forma de listas y matrices como listas de vectores sobre un anillo. Los conceptos sobre el método de Gauss-Jordan se encuentran en los apuntes de teoría "Sistemas de ecuaciones lineales" del Departamento de Álgebra ([4]).

- Finalmente, incorporamos la teoría sobre anillos coherentes en Haskell. El objetivo de este capítulo es poder resolver sistemas de ecuaciones de la forma $M\vec{X} = \vec{b}$ en anillos fuertemente discretos y coherentes. La primera sección del capítulo

contiene la noción de anillo fuertemente discreto. En la siguiente sección, se define la noción de anillo coherente y se demuestran unas proposiciones que nos permiten implementar la resolución de sistemas de ecuaciones homogéneos. Al final del capítulo, trabajamos con anillos fuertemente discretos y coherentes, los cuáles nos permiten resolver sistemas de ecuaciones no homogéneos. Para el desarrollo de este capítulo, se ha hecho uso de las notas de teoría "Coherent Ring" de Thierry Coquand ([3]) y para el código, se ha utilizado la tesis de Anders Mörtberg ([6]).

Para poder llevar acabo este trabajo he necesitado utilizar varias herramientas informaticas, daré un breve resumen de cada una de ellas:

- **Ubuntu**
  Es un sistema operativo de código abierto para ordenadores. Es una distribución de Linux basada en la arquitectura de Debian. Actualmente corre en ordenadores de escritorio y servidores, en arquitecturas Intel, AMD y ARM. Está orientado al usuario promedio, con un fuerte enfoque en la facilidad de uso y en mejorar la experiencia del usuario. Está compuesto de múltiple software normalmente distribuido bajo una licencia libre o de código abierto. Para desarrollar el trabajo hemos utilizado la versión Ubuntu 16.04 que se encuentra en la web oficial de Ubuntu, los pasos para su instalación están en el apéndice A.

- **LaTeX**
  Es un sistema de composición de textos, orientado a la creación de documentos escritos que presenten una alta calidad tipográfica. Por sus características y posibilidades, se usa en la generación de artículos y libros científicos para incluir expresiones matemáticas, entre otros. LaTeX está formado por un gran conjunto de macros de TeX, con la intención de facilitar el uso del lenguaje de composición tipográfica TeX. LaTeX es software libre bajo licencia LPPL.

- **Haskell**
  Es un lenguaje de programación puramente funcional con semánticas no estrictas y fuertemente tipado. Dentro de Haskell, encontramos **haskell literario**, con el cuál podemos escribir código en el lenguaje de programación de Haskell al mismo tiempo que podemos redactar texto con las utilidades que ofrece LaTeX. Todo el trabajo está escrito en **haskell literario**. Haskell 8.0.2 es la versión que se ha utilizado para realizar el trabajo.

- **Emacs**
  Es un editor de texto libre, extensible y personalizable. Emacs es el editor que he usado para programar en Haskell. Este dispone de muchas abreviaciones de te-

clado para facilitar su uso de una forma más rápida, las más utilizadas se pueden ver en el apéndice B.

- **GitHub**
  Es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas. El código de los proyectos alojados en GitHub se almacena de forma pública, aunque utilizando una cuenta de pago, también permite tener repositorios privados. En mi perfil de GitHub se encuentra el repositorio con el trabajo. Gracias a esta plataforma puedo compartir los avances con mi tutora. Esto es posible porque al crear un repositorio de GitHub, se pueden incluir colaboradores, que podrán añadir y modificar los elementos del repositorio. Para poder subir los archivos a GitHub podemos hacerlo desde la terminal de Ubuntu, los pasos a seguir están en el apéndice C.

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]). Las funciones predefinidas en Haskell que vamos a utilizar en los siguientes capítulos pertenecen a la librería `Data.List` de Haskell.

Con la programación puramente funcional no ejecutamos órdenes, sino más bien, definimos como son las cosas. Lo único que puede hacer una función es calcular y devolver algo como resultado. Al definir una función se le asocia un tipo que determina el comportamiento de la función, esto permite al compilador razonar acerca de el comportamiento, y nos permite ver si una función es correcta o no antes de ejecutarla. De esta forma podemos construir funciones más complejas uniendo funciones simples.

Haskell es un lenguaje tipificado estáticamente. Esto significa que el compilador sabe que trozos del código se corresponde con números enteros, con cadenas de texto, etc. Gracias a esto muchos de los errores los detectamos en tiempo de compilación.

Vamos a explicar brevemente los conceptos importantes para entender mejor la programación funcional que utiliza Haskell:

- **Funciones de orden superior**

  Las funciones de Haskell pueden tomar funciones como parámetros y devolver funciones como resultado. Una función que hace ambas cosas o alguna de ellas se llama función de orden superior. Es decir, si una función toma como argu-

mento a otra función o devuelve una función como resultado se dice que es una función de orden superior. Por ejemplo, las funciones `foldr`, `filter` y `zipWith` son funciones de orden superior, todas ellas toman a otra función como argumento.

- **Tipos**

  Cada función en Haskell tiene asociada un tipo. Al programar la función no es necesario especificar el tipo, pues Haskell lo identifica solo. Es conveniente especificar el tipo sobre el que queremos utilizar la función, para que al crear nuevas funciones no tengamos problemas de tipos, o que los problemas sean más faciles de identificar.

  Un tipo en Haskell es como un conjunto de elementos que tienen algo en común o una categoría. Por ejemplo el grupo de los números enteros Z se representa por el tipo `Int`. Los tipos más conocidos de Haskell son: `Integer`, `Int`, `Char`, `Bool`, `Double` y `Float`.

- **Clases de tipos**

  Las clases de tipos son una especie de interfaz que define algún tipo de comportamiento. Si un tipo es miembro de una clase de tipos, significa que ese tipo soporta e implementa el comportamiento que define la clase de tipos. Cada clase tiene unas propiedades que se aplican a los elementos que se utilizan estando en dicha clase. Por ejemplo: `Eq`, `Show` y `Ord` son las clases más conocidas.

  Podemos crear nuevas clases de tipos escribiendo `class Ring a where`, la nueva clase de tipo que hemos definido se llama `Ring`. La `a` es la variable de tipo, esta representará el tipo sobre el que se trabajará en la clase `Ring`. No tiene porque llamarse `a`, solo debe ser una palabra en minúsculas. Debajo del `where` definimos varias funciones. No es obligatorio implementar los cuerpos de las funciones, solo debemos especificar las declaraciones de tipo de las funciones.

- **Restricciones de clases de tipos**

  Para concretar sobre que propiedades queremos trabajar se puede restringir una clase dentro de otra. Por ejemplo, la función `group` tiene el siguiente tipo:

```
group :: Eq a =>[a] ->[[a]]
```

Aquí vemos que `group` tiene la propiedad de la igualdad `==`, porque la hereda de la clase `Eq` porque con el símbolo `=>` es el que indica que estamos restringiendo a una clase. Sin embargo la función `elem` con el siguiente tipo:

```
elem :: (Foldable t, Eq a) =>a ->t a ->Bool
```

Esta restringida a dos clases, de esta forma la función `elem` tiene las propiedades de ambas clases.

- **Instancias**

  Un tipo puede ser una instancia de una clase si soporta el comportamiento de la clase. Por ejemplo: El tipo `Int` es una instancia de la clase `Eq`, ya que la clase de tipos`Eq` define el comportamiento de cosas que se pueden equiparar, y los números enteros pueden equipararse.

  También podemos utilizar instancias sobre las clases que hayamos creado. Al declarar una instancia, estamos declarando el tipo sobre el que la clase va a actuar bajo los atributos que se definan. Las funciones que se definan y estén restringidas a la nueva clase, verificará los atributos de esta clase.

Capítulo 2

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.

2.1. 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 2.1.1. *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 conmutativa. Esta función al igual que la anterior de potencia recibe un elemento de la clase `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."
```

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

Definición 2.2.1. *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
```

2.3. 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 2.3.1. *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
```

Un ejemplo de dominio de integridad 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 dominio de integridad, comprueba que se verifiquen las operaciones necesarias para ser un dominio de integridad.

```
instance IntegralDomain Integer
```

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 2.3.2. *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
```

```
-- | El anillo de los reales con las operaciones usuales es cuerpo:

instance Ring Double where
     (<+>)  = (+)
     (<**>) = (*)
     neg a  = 1/a
     zero   = 0
     one    = 1

instance CommutRing Double

instance IntegralDomain Double

instance Field Double where
    inv a = 1/a
```

2.4. 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 2.4.1. *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 2.4.2. *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 canónicos, 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 2.4.3. *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 2.4.4. *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 2.4.5. *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 3

Vectores y Matrices

En este capítulo implementaremos los vectores y matrices como listas para poder usarlas en 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 anillos sobre cualquier tipo a.

3.1. 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 módulo `TAHVector`

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

import Control.Monad (liftM)

import Test.QuickCheck
import TAHField
```

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. Daremos más detalle cuando la utilicemos.

Comenzamos por implementar las nociones básicas de los vectores. Creamos un

nuevo tipo para definir un vector, usaremos las listas para trabajr con los vectores en Haskell. 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` generamos un tipo dado de forma aleatoria en nuestro caso del tipo vector.

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
```

Para trabajar con vectores, haremos uso de la clase de tipos `Functor` de Haskell. 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`, *ynoproporcionaningunaimplementacinpordefecto.Eltipode*`fmap`*essimilar*

En nuestro caso crearemos una instancia con la clase de tipos `Functor` sobre el constructor `Vector` con el objetivo de aplicar una función `f` sobre un vector. Definiremos la función `(fmap f)` de forma que devuelva un vector con la función `f` aplicada a cada componente del vector.

```
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
```

3.2. 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 vectores de forma

que cada vector será una fila de la matriz. 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`

```
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
```

Antes de comenzar, hemos importado 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.

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)
```

Una vez implementado el tipo de las matrices vamos a definir la función para construir una matriz de dimensión $m \times n$ 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 una 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 el elemento en la posición (i, j).

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

-- Ejemplo:
-- > unM (M [Vec [2,3], Vec [3,4]])
--    [[2,3],[3,4]]
-- | Aplicamos la función 'unM' a cada vector de la lista.
unMVec :: Matrix r -> [[r]]
unMVec = map unVec . unM
-- Ejemplo:
-- > unMVec (M [Vec [2,3], Vec [3,4]])
--    [[2,3],[3,4]]

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

```
-- Ejemplo:
-- > vectorToMatrix (Vec [2,3,4])
--    [2,3,4]

-- | 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"
-- Ejemplos:
-- > matrixToVector (M [Vec [2,3], Vec [4,5]])
--    *** Exception: No pueden tener dimensiones distintas
-- > matrixToVector (M [Vec [2,3]])
--    [2,3]

-- | Obtener el elemento de la posición (i+1,j+1).
(!!!) :: Matrix a -> (Int,Int) -> a
m !!! (i,j) | i >= 0 && i < rows && j >= 0 && j < cols = unMVec m !! i !! j
            | otherwise = error "!!!: Fuera de los límetes"
  where
  (rows,cols) = dimension m

-- Ejemplos:
-- > (M [Vec [2,3,4], Vec [4,5,6], Vec [7,8,9]]) !!! (2,3)
--    *** Exception: !!!: Fuera de los límetes
-- > (M [Vec [2,3,4], Vec [4,5,6], Vec [7,8,9]]) !!! (1,2)
--    6
-- > (M [Vec [2,3,4], Vec [4,5,6], Vec [7,8,9]]) !!! (0,1)
--    3
```

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 que son 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)))
-- Ejemplo:
-- > dimension (M [Vec [2,3,4], Vec [4,5,6], Vec [7,8,9]])
--    (3,3)

-- | Comprobar si una matriz es cuadrada.
```

```
isSquareMatrix :: Matrix r -> Bool
isSquareMatrix (M xs) = all (== l) (map lengthVec xs)
                      where l = length xs
-- Ejemplo:
-- > isSquareMatrix (M [Vec [2,3,4], Vec [4,5,6], Vec [7,8,9]])
--    True
-- > isSquareMatrix (M [Vec [2,3,4], Vec [4,5,6]])
--    False

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

-- Ejemplo:
-- > transpose (M [Vec [2,3,4], Vec [4,5,6]])
--    [2,4]
--    [3,5]
--    [4,6]
```

Recordamos que la suma de matrices 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 a la fila y columna. Aquí si podemos utilizar la operación `mulVec` entre vectores pues devuelve un valor que no es un vector, por tanto obtenemos una lista de vectores. 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))

-- Ejemplo:
-- > addM (M [Vec [2,3,4], Vec [4,5,6]]) (M [Vec [1,0,2], Vec [1,2,3]])
--    [3,3,6]
--    [5,7,9]
```

```
-- > addM (M [Vec [2,3,4], Vec [4,5,6]]) (M [Vec [1,0,2]])
--    *** Exception: Las dimensiones de las matrices son distintas

-- | 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) ]

-- Ejemplo:
-- >  mulM (M [Vec [2,3], Vec [4,5]]) (M [Vec [1,0,2], Vec [1,2,3]])
--     [5,6,13]
--     [9,10,23]
-- > mulM (M [Vec [2,3,4], Vec [4,5,6]]) (M [Vec [1,0,2], Vec [1,2,3]])
--    *** Exception: La dimensión de colunmas y filas es distinta
```

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
```

Veamos un ejemplo sobre los números enteros de `(fmap f)`, sumar 2 a una matriz:

```
-- > fmap (<+> 2) (M [Vec [2,3], Vec [4,5]])
--    [4,5]
--    [6,7]
```

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)

-- Ejemplo:
-- > identity 3
--    [1,0,0]
```

```
--    [0,1,0]
--    [0,0,1]

-- 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, al igual que hicimos con los vectores. Realizaremos operaciones entre filas y columnas, y veremos que estas operaciones no afectan a la dimensión de la matriz. Hacemos esta restricción debido a en el siguiente capítulo necesitaremos estas operaciones con matrices y estaremos 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 tenga la fila que queramos modificar 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

-- Ejemplo:
-- > scaleMatrix (M [Vec [2,3], Vec [4,5]]) 1 5
--    [2,3]
--    [20,25]

-- 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)

-- Ejemplo:
-- > swap (M [Vec [2,3], Vec [4,5], Vec [6,7]]) 0 1
--    [4,5]
--    [2,3]
--    [6,7]

-- 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

-- | Al realizar dos veces un mismo intercambio volvemos a la matriz
--   de inicio.
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@(Vec xs) x)` realizamos la operación de su-

mar un vector (`row@(Vec xs)`) 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

-- Ejemplo:
-- > addRow (M [Vec [2,3], Vec [4,5], Vec [6,7]]) (Vec [10,15]) 1
--    [2,3]
--    [14,20]
--    [6,7]

-- 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

-- Ejemplo:
-- > addCol (M [Vec [2,3], Vec [4,5], Vec [6,7]]) (Vec [10,15,10]) 1
--    [2,13]
--    [4,20]
--    [6,17]
```

Finalmente, realizaremos la operación anterior de sumar un vector a una fila o columna 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

-- Ejemplo:
-- > subRow (M [Vec [2,3], Vec [4,5], Vec [6,7]]) (Vec [10,15,10]) 1
--    [2,3]
--    [-6,-10]
--    [6,7]
-- > subCol (M [Vec [2,3], Vec [4,5], Vec [6,7]]) (Vec [10,15,10]) 1
--    [2,-7]
--    [4,-10]
--    [6,-3]
```

Gracias a todo lo anterior ahora podemos implementar el método de Gauss-Jordan para poder resolver sistemas $A\vec{x} = \vec{b}$ donde A es una matriz mxn y $\vec{b}$ es un vector columna de n filas.

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 aplicar el método de Gauss o escalonar una matriz.

Para empezar damos las funciones para obtener un 0 en las posiciones de debajo del pivote y la segunda función consiste en localizar el siguiente pivote, comenzando la búsqueda desde una entrada fijada de la matriz. Por ejemplo, dada una matriz 3×3 `findPivot m (0,1)`, nos devolverá el primer cero que aparezca en la columna 2 comenzando desde la fila 2, es decir, mirará las posiciones `(2,2)` y `(2,3)`. Recordemos que en el algoritmo el índice empieza en 0, por lo que miraría las posiciones `(1,1)` y `(1,2)`.

```
-- | Multiplicar por el escalar s 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
```

```
-- Ejemplo:
-- > pivot (M [Vec [2,3,4], Vec [4,5,6], Vec [4,8,9]]) (-2) 0 1
--    [2,3,4]
--    [0,-1,-2]
--    [4,8,9]

-- | Encontrar el primer cero que aparezca la columna c empezando
--   desde la fila r y devolve el valor del pivote 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

-- Ejemplos:
-- > findPivot (M [Vec [1,3,4], Vec [0,5,6], Vec [0,8,9]]) (0,0)
--    Nothing
-- Devuelve Nothing porque en la primera columna y la primera fila no hay 0
-- > findPivot (M [Vec [1,3,4], Vec [0,5,6], Vec [0,8,9]]) (0,1)
--    Just (5,1)
-- Devuelve (5,1) porque 5 es el primer valor distinto de cero comenzando
-- desde la fila 1  (sin contar la propia fila 1) hacia abajo siguiendo
-- la columna 2.
-- > findPivot (M [Vec [1,3,4], Vec [0,0,6], Vec [0,8,9]]) (0,1)
--    Just (8,2)
-- Al colocar un 0 en la posición donde antes teníamos un 5 vemos como ahora
-- nos devuelve el primer valor distinto de cero que aparezca en la columna
-- 2 empezando desde la fila 1 (sin contar la propia fila 1)
```

Con la siguiente función buscamos escalonar la matriz de forma que todo lo que quede debajo de la diagonal sean ceros. 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 se corresponde con el inverso del pivote para que al sumarlo obtengamos un cero.

```
-- | Escalonar la matriz.
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)
  -- Con cancelOut hacemos cero en las posiciones de debajo del pivote.
  xs = unMVec m

--Ejemplos:
-->  fE (M [Vec [2,3,4], Vec [4,5,6], Vec [7,8,9]])
--[2.0,3.0,4.0]
--[0.0,6.5,8.0]
--[0.0,0.0,16.013824884792626]

--> fE (M [Vec [1,3,4], Vec [0,0,6], Vec [0,8,9]])
--[1.0,3.0,4.0]
--[0.0,8.0,9.0]
--[0.0,0.0,6.0]

--> fE (M [Vec [1,0,2], Vec [2,-1,3], Vec [4,1,8]])
--[1.0,0.0,2.0]
--[0.0,-1.0,4.0]
--[0.0,0.0,4.5]
```

Para calcular la forma escalonada para resolver un sistema $A\vec{x} = \vec{b}$, seguimos necesitando que los elementos de las matrices pertenezca a un cuerpo. Primero aplicamos Gauss, es decir, obtenemos ceros por debajo de la diagonal. Aplicando las operaciones al vector $\vec{b}$ también. De esta forma se queda el sistema preparado para resolver de abajo a arriba cada incógnita. Además, con esta función dejamos los pivotes con unos, para facilitar la solución del sistema.

```
-- | 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)
      -- Si estamos en la última fila o columna de la matriz
    | c == mc || r == mr =
      -- Descompone la matriz en A y b de nuevo.
```

```
      (matrix *** Vec) $ unzip $ map (init &&& last) $ unMVec m

      -- Si hay un cero en la posición (r,c) entonces intercambiamos la
      -- fila por la primera fila con elemento no nulo en la columna
      -- del pivote.
    | m !!! rc == zero   = case findPivot m rc of
      Just (_,r') -> fE (swap m r r') rc

      -- Si todos los elementos de la columna pivote 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 pivote.
      Just (v,r') -> fE (pivot m (neg v) r r') (r,c)
      -- Si todos los elementos en la columna pivote 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

  -- Forma la matriz A añadiendole la columna b.
  m' = matrix $ [ r ++ [x] | (r,x) <- zip (unMVec m) (unVec v) ]

--Ejemplos:
--> forwardElim (M [Vec [1,3,4], Vec [0,0,6], Vec [0,8,9]],Vec [4,5,6])
-- ([1.0,3.0,4.0]
-- [0.0,1.0,1.125]
-- [0.0,0.0,1.0]
-- ,[4.0,0.75,0.8333333333333333])
```

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 cada 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]
--Ejemplos:
--> jordan (M [Vec [1,3,4], Vec [0,1,1.125], Vec [0,0,1]],Vec [4,0.75,0.84])
-- ([1.0,0.0,0.0]
-- [0.0,1.0,0.0]
-- [0.0,0.0,1.0]
-- ,[4.481964329257672,1.8082010582010584,0.84])
--
--> jordan (M [Vec [1,3,4], Vec [1,0,6], Vec [0,8,9]],Vec [4,5,6])
-- ([1.0,0.0,0.0]
-- [1.0,0.0,0.0]
-- [0.0,8.0,9.0]
-- ,[4.107965009208104,5.027777777777778,6.0])
```

Finalmente, podemos realizar el método de Gauss-Jordan, con el objetivo de resolver un sistema de ecuaciones de la forma $A\vec{x} = \vec{b}$. Primero obtenemos la matriz con solo la diagonal que es la obtenida tras aplicar Gauss-Jordan esto lo obtenemos con la función que denotaremos `gaussElim`.

La última función que denotaremos `gaussElimCorrect`, recibe la partición de la matriz A y el vector columna $\vec{b}$. Con ella comprobamos dos cosas. La primera que la dimensión de las filas de la matriz A coincida con la dimensión del vector $\vec{b}$ y que A sea una matriz cuadrada. Una vez se cumple esto lo siguiente que comprueba es que el vector que se obtiene de `gaussElim` al multiplicarlo por la matriz A coincide con el vector $\vec{b}$.

```
-- | 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) -> Property
gaussElimCorrect m@(a,b) =
  fst (dimension a) == lengthVec b && isSquareMatrix a ==>
  matrixToVector (transpose
```

```
  (a `mulM` transpose (M [snd (gaussElim m)])) == b
```

Capítulo 4

Anillos Coherentes

El objetivo de este capítulo será ver cuando un anillo es coherente y fuertemente discreto. Gracias a esto podremos resolver sistemas de ecuaciones sobre estos anillos. Antes de comenzar, introduciremos la noción de anillo fuertemente discreto.

4.1. 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 4.1.1. *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 no es posible decidir si dos números irracionales son iguales en tiempo finito.

Definición 4.1.2. *Un anillo es fuertemente discreto si la pertenencia a un ideal es decidible.*

Para introducir este concepto definimos una clase restringida a la clase de tipo *Ring*:

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

El objetivo de este método es decidir si un elemento del anillo pertene al ideal, por ello hacemos uso del constructor `(Maybe [a])`. En el caso de que no pertenezca al ideal, `member` devolverá `Nothing`. Por otro lado, si un elemento pertenece a un ideal, significa que podemos escribir dicho elemento mediante una combinación lineal de los generadores del ideal. Por ello, si el elemento pertenece al ideal, `member` nos devolverá la lista con los coeficientes de los generadores del ideal al que nuestro elemento pertenece.

Para verificar que una especificación concreta de `member` es correcta definimos una función que denotaremos `(propStronglyDiscrete x id@(Id xs))`, esta devolverá un booleano, `True` cuando `member` haya funcionado bien y `False` cuando no haya devuelto lo esperado. En caso de que no pertenezca al ideal y devuelva `Nothing` significa que funciona correctamente luego obtendremos un `True`. Si `x` pertenece al ideal generado por `xs` entonces comprobará que la lista de coeficientes que `member` ha devuelto al multiplicarla por la lista de generadores del ideal, `xs`, la suma resultante es `x` y entonces devolverá un `True`.

```
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
```

4.2. 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
  ) where
```

```
import Test.QuickCheck

import TAHIntegralDomain
import TAHIdeal
import TAHStronglyDiscrete
import TAHVector
import TAHMatrix
```

Definición 4.2.1. *Un anillo R es coherente si dado un vector $\vec{m} \in R^{1\times n}$ existe una matriz $L \in R^{n\times r}$ para $r \in \mathbb{N}$ tal que $\vec{m}L = \vec{0}$ y*

$$\vec{m}\vec{X} = 0 \Leftrightarrow \exists\ \vec{Y} \in R^{r\times 1} \text{ tal que } \vec{X} = L\vec{Y}$$

esto es,

$$\begin{pmatrix} m_1 & m_2 & \cdots & m_n \end{pmatrix} \begin{pmatrix} l_{11} & l_{12} & \cdots & l_{1r} \\ l_{21} & l_{22} & \cdots & l_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nr} \end{pmatrix} = \begin{pmatrix} 0 & \cdots & 0 \end{pmatrix}_{1\times r} \quad y$$

$$\begin{pmatrix} m_1 & m_2 & \cdots & m_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = 0 \Leftrightarrow \exists \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_r \end{pmatrix} \quad tal\ que$$

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} l_{11} & l_{12} & \cdots & l_{1r} \\ l_{21} & l_{22} & \cdots & l_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nr} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_r \end{pmatrix}$$

De esta forma es posible calcular las soluciones de un sistema de ecuaciones en un anillo coherente. El vector $\vec{m}$ es solución del sistema homogéneo y la matriz L nos proporciona un conjunto de generadores finito de las soluciones del sistema. De esta forma, podemos obtener todas las posibles soluciones del sistema.

La propiedad de la coherencia es bastante difícil de implementar en Haskell. El con-

tenido que se puede implementar es que es posible calcular la matriz L dada $\vec{m}$ tal que $\vec{m}L = 0$.

En esta sección todos los anillos de los que partimos son dominios de integridad, por lo que al construir la clase de los anillos coherentes haremos una restricción a la clase de *IntegralDomain*. Introducimos la clase de anillos coherentes:

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

Al igual que ocurría con `member` en el anterior capítulo, aquí *solve* es una función que no tiene una definición concreta. El objetivo de esta función es que dado un vector $\vec{m} \in R^n$ devuelva una matriz $L \in R^{n\times r}$ de forma que al multiplicar ambos el vector resultante sea un vector fila de ceros.

Para verificar que una definición concreta de `solve` es correcta especificamos unas funciones para realizar dicha comprobación. La función que denotaremos `propCoherent` es la encargada de comprobar que la multiplicación de $\vec{m}$ por L sea nula. Para ello, se ayuda de una segunda función que denotaremos por `isSolution`, esta comprueba que el vector que se obtiene tras la multiplicación de $\vec{m}L$ es un vector de ceros.

```
-- | Test para comprobar que la multiplicación del vector M por la matriz
--   encontrada por solve (la matriz L) sea un vector de ceros.
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)
```

Empezaremos por resolver sistemas de ecuaciones homogéneos sobre un anillo coherente. Nuestro objetivo es encontrar todas las posibles soluciones del sistema homogéneo, solo que esta vez tenemos una matriz M y no un vector.

Proposición 4.2.1. *En un anillo coherente R es posible resolver un sistema $M\vec{X} = \vec{0}$ donde*

$M \in R^{r\times n}$ y $\vec{X} \in R^{n\times 1}$. Es decir,

$$\begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{r1} & m_{r2} & \cdots & m_{rn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}_{r\times 1}$$

Prueba 4.2.1. *Sean $\vec{m}_i \in R^{1\times n}$, $\vec{m}_i = \begin{pmatrix} m_{i1} & \cdots & m_{in} \end{pmatrix}$ las filas de M.*
Por coherencia es posible resolver $\vec{m}_1\vec{X} = 0$ y obtener un $L_1 \in R^{n\times p_1}$ tal que

$$\begin{pmatrix} m_{11} & \cdots & m_{1n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = 0 \Leftrightarrow \exists \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{p_1} \end{pmatrix} \text{ tal que}$$

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} l_{1(11)} & l_{1(12)} & \cdots & l_{1(1p_1)} \\ l_{1(21)} & l_{1(22)} & \cdots & l_{1(2p_1)} \\ \vdots & \vdots & \ddots & \vdots \\ l_{1(n1)} & l_{1(n2)} & \cdots & l_{1(np_1)} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{p_1} \end{pmatrix}$$

Si imponemos, $\vec{m}_2\vec{X} = 0$, como $\vec{X} = L_1\vec{Y} \Rightarrow \vec{m}_2L_1\vec{Y} = 0$.
Por coherencia sobre $\vec{Y}$ existe $L_2 \in R^{p_1\times p_2}$ tal que $\vec{m}_2L_1\vec{Y} = 0 \Leftrightarrow \exists\, \vec{Z} \in R^{p_2\times 1}$ tal que $\vec{Y} = L_2\vec{Z}$.
Por tanto, nos queda que

$$\begin{cases} \vec{X} = L_{1(n\times p_1)}\vec{Y}_{(p_1\times 1)} \\ \vec{Y} = L_{2(p_1\times p_2)}\vec{Z}_{(p_2\times 1)} \end{cases} \Rightarrow \vec{X} = L_{1(n\times p_1)}L_{2(p_1\times p_2)}\vec{Z}_{(p_2\times 1)}$$

Iterando este método la solución $\vec{X} = L_1L_2\cdots L_r\vec{Z}$ con $L_i \in R^{p_{i-1}\times p_i}$, $p_0 = n$ y $\vec{Z} \in R^{p_m\times 1}$ puede ser calculada.

La proposición anterior nos muestra la forma de resolver mediante recursión un sistema $M\vec{X} = \vec{0}$, veamos como hacerlo en Haskell. Siguiendo la prueba de la proposición anterior, comenzamos a aplicar coherencia con la primera fila de la matriz M y así vamos obteniendo por coherencia una nueva matriz en cada iteración hasta obtener la solución del sistema de ecuaciones. Mediante la función `solveMxN` calcula la matriz obtenida por recursión $L_1L_2\cdots L_r$. Con una segunda función, que denotaremos `propSolveMxN` comprobaremos que la matriz obtenida por `solveMxN` al multiplicarla

por la matriz dada M es solución del sistema homogéneo es decir el resultado de la multiplicación es un vector de ceros.

```
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 esta especificación de solve devuelve
--  un vector de ceros.
propSolveMxN :: (Coherent a, Eq a) => Matrix a -> Bool
propSolveMxN m = isSolution m (solveMxN m)
```

Ahora consideraremos la intersección de dos ideales finitamente generados en un anillo coherente. Esto es necesario para poder resolver más adelante sistemas utilizando la intersección entre ideales.

Proposición 4.2.2. *La intersección de dos ideales finitamente generados en un anillo coherente R está finitamente generada.*

Prueba 4.2.2. *Sean $I =< a_1, \cdots, a_n >$ y $J =< b_1, \cdots, b_m >$ dos ideales finitamente generados en R. Consideramos el sistema $AX - BY = 0$, donde $A = (a_1 \cdots a_n)$ y $B = (b_1 \cdots b_m)$ son vectores filas.*

Como el anillo es coherente, entonces es posible calcular un número finito de generadores $\{(X_1, Y_1), \cdots, (X_p, Y_p)\}$ de la solución.

Esto es,

$$\begin{aligned} AX_1 &= BY_1 \\ &\vdots \\ AX_p &= BY_p \end{aligned}$$

y si $(x_1, \cdots, x_n), (y_1, \cdots, y_m)$ verifican

$$A \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = B \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \tag{4.2}$$

Entonces se tiene,

$$\exists\ \lambda_1, \cdots, \lambda_p\ :\ \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \lambda_1 X_1 + \cdots + \lambda_p X_p \tag{4.3}$$

$$\exists\ \mu_1, \cdots, \mu_p\ :\ \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = \mu_1 Y_1 + \cdots + \mu_p Y_p \tag{4.4}$$

Si $z \in I \cap J \Rightarrow \exists\ \alpha_1, \cdots, \alpha_n, \beta_1, \cdots, \beta_m \in R$ *tales que*

$$z = \alpha_1 a_1 + \cdots + \alpha_n a_n = \beta_1 b_1 + \cdots + \beta_m b_m$$

$$\Rightarrow\ (a_1, \cdots, a_n) \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = (b_1, \cdots, b_m) \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_m \end{pmatrix}$$

son soluciones del sistema (4.2).

De (4.2),(4.3) y (4.4) se tiene que

$$(a_1, \cdots, a_n)\left(\lambda_1 X_1 + \cdots + \lambda_p X_p\right) = (b_1, \cdots, b_m)\left(\mu_1 Y_1 + \cdots + \mu_p Y_p\right)$$

$$= \lambda_1 A X_1 + \cdots + \lambda_p A X_p = \mu_1 B Y_1 + \cdots + \mu_p B Y_p$$

Por tanto, un conjunto de generadores para la intersección es $\{AX_1, \cdots, AX_p\}$ *y otro conjunto de generadores es* $\{BY_1, \cdots, BY_p\}$.

A continuación daremos una proposición importante en la teoría de anillos coherentes, con ella podemos probar si un anillo R es coherente.

Proposición 4.2.3. *Si R es un dominio de integridad tal que la intersección de dos ideales finitamente generados está finitamente generada, entonces R es coherente.*

Prueba 4.2.3. *Lo probaremos mediante inducción en la longitud del sistema a resolver. Primero consideramos* $ax = 0$. *Aquí la solución es trivial, pues R es un dominio de integridad, por lo que* $a \neq 0$, *por tanto se tiene que* $x = 0$. *Suponemos cierto que es posible resolver un sistema en* $(n-1)$ *variables y consideramos el caso con* $n \geq 2$ *variables:*

$$a_1 x_1 + \cdots + a_n x_n = 0$$

Si $a_1 = 0$, *un conjunto de soluciones del sistema está generado por* $(1, 0, \cdots, 0)$, *pero también es posible usar la hipótesis de inducción y obtener los generadores* $\{(v_{i2}, \cdots, v_{in})\}$ *para el sistema con* $x_2, \cdots, x_n$ *y las soluciones del sistema con* n *incógnitas están generadas por* $\{(1, 0, \cdots, 0), (0, v_{12}, \cdots, v_{1n}), \cdots, (0, v_{s2}, \cdots, v_{sn})\}$.

Si $a_1 \neq 0$, *el conjunto de generadores* $\{(0, v_{12}, \cdots, v_{1n}), \cdots, (0, v_{s2}, \cdots, v_{sn})\}$ *de las soluciones puede obtenerse también por hipótesis de inducción. Además, por hipótesis es posible encontrar* $t_1, \cdots, t_p$ *tales que*

$$< a_1 > \cap < -a_2, \cdots, -a_n > = < t_1, \cdots, t_p >$$

donde $t_i = a_1 w_{i1} = -a_2 w_{i2} - \cdots - a_n w_{in}$.

Si $x_1, \cdots, x_n$ *es solución* $\Rightarrow a_1 x_1 + \cdots + a_n x_n = 0 \Rightarrow a_1 x_1 = -a_2 x_2 - \cdots - a_n x_n \Rightarrow$

$$\begin{cases} a_1 x_1 \in < t_1, \cdots, t_p > \quad y \\ -a_2 x_2 - \cdots - a_n x_n \in < t_1, \cdots, t_p > \end{cases}$$

Por lo que existen unos u_i *tales que*

$$a_1 x_1 = -a_2 x_2, \cdots, -a_n x_n = \sum_{i=1}^{p} u_i t_i$$

Por tanto se tiene que

$$a_1 x_1 = \sum_{i=1}^{p} u_i t_i = \sum_{i=1}^{p} u_i a_1 w_{i1} \Rightarrow x_1 = \sum_{i=1}^{p} u_i w_{i1}$$

Esta implicación podemos hacerla ya que $a \neq 0$ *y* R *es dominio de integridad. De forma análoga,*

$$-a_2 x_2 - \cdots - a_n x_n = \sum_{i=1}^{p} u_i t_i = \sum_{i=1}^{p} u_i (-a_2 w_{i2} - \cdots - a_n w_{in})$$

Reorganizando nos queda

$a_2(x_2 - \sum_{i=1}^{p} u_i w_{i2}) + \cdots + a_n(x_n - \sum_{i=1}^{p} u_i w_{in}) = 0 \Rightarrow$

$\left(x_2 - \sum_{i=1}^{p} u_i w_{i2}, \cdots, x_n - \sum_{i=1}^{p} u_i w_{in}\right)$ *es combinación lineal de*

$\{(0, v_{12}, \cdots, v_{1n}), \cdots, (0, v_{s2}, \cdots, v_{sn})\}$ *que recordamos que son los generadores*

de las soluciones de $a_2 x_2 + \cdots + a_n x_n = 0$

Finalmente, tenemos que existen unos coeficientes α_i *con* $i = 1, \cdots, s$ *tales que podemos escribir lo anterior como*

$$x_1 = \sum_{i=1}^{p} u_i w_{i1}$$

$$x_2 = \sum_{i=1}^{p} u_i w_{i2} + \alpha_1 v_{12} + \cdots + \alpha_s v_{s2}$$

$$\vdots$$

$$x_n = \sum_{i=1}^{p} u_i w_{in} + \alpha_1 v_{1n} + \cdots + \alpha_s v_{sn}$$

Esto es,

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = u_1 \begin{pmatrix} w_1 1 \\ w_1 2 \\ \vdots \\ w_1 n \end{pmatrix} + \cdots + u_p \begin{pmatrix} w_p 1 \\ w_p 2 \\ \vdots \\ w_p n \end{pmatrix} + \alpha_1 \begin{pmatrix} 0 \\ v_1 2 \\ \vdots \\ v_1 n \end{pmatrix} + \cdots + \alpha_s \begin{pmatrix} 0 \\ v_s 2 \\ \vdots \\ v_s n \end{pmatrix}$$

Luego, obtenemos $\{(w_{11}, \cdots, w_{1n}), \cdots, (w_{r1}, \cdots, w_{rn})\}$ *y* $\{(0, v_{12}, \cdots, v_{1n}), \cdots, (0, v_{s2}, \cdots, v_{sn})\}$ *que generan el conjunto de las soluciones.*

Esto proporciona un método para probar que los anillos son coherentes. Ahora vamos a ver cómo calcular la intersección de los ideales finitamente generados. Esto implicará que el anillo es coherente. También muestra que la coherencia de los anillos se puede caracterizar solo en términos de la intersección finita de ideales finitamente generados.

Vamos a dar un algoritmo para obtener una solución del sistema mediante la intersección, basándonos en las propocisión anterior.

```
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
  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:) $ solveInt xs)
  -- Aquí si x=0 tenemos el primer generador (1,0,..,0) como primer
  -- elemento de la matriz que devuelve.
  -- Con map resolvemos el resto de xs añadiendo delante 0, para
  -- conseguir los generadores (0,v_i1,..,v_in). Finalmente nos
  -- queda: [[1,0,..,0],[0,v_11,..,v_1n],[0,v_s1,..,v_sn]]

| isSameIdeal int as bs = s ++ m'
  --Este es el caso x /= 0, aquí resolvemos por intersección

| otherwise = error "solveInt: No se puede calcular la intersección"
  where
  as = Id [x]              -- a_1x_1
  bs = Id (map neg xs)     -- -a_2x_2-..-a_nx_n

  -- 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
```

La función `(solveWithIntersection (Vec xs) int)` recibe como argumento de entrada el vector a resolver $\vec{X}$ así como la intersección de dos ideales finitamente generados en forma de terna `(Ideal a,[[a]],[[a]])` de forma que `(Ideal a)` son los generadores del ideal intersección, las otras dos listas de listas contienen los coeficientes correspondiente a cada uno de los dos ideales de los que se obtiene la intersección. Es decir, como `(Ideal a)` es el resultado de interseccionar estos dos ideales, si un elemento pertenece a la intersección, este puede escribirse como combinación lineal de cada uno de los dos ideales.

Para el caso en el que $a_1 \neq 0$, aplicamos `(isSameIdeal int as bs)`. Recordamos que esta función devuelve un booleano cuando se verifica lo comentado anteriormente. De esta forma, si `isSameIdeal` devuelve `True` se realiza la intersección de los generadores que se obtienen de la intersección de los dos ideales, junto con los generadores de la solución obtenida al resolver $a_2x_2 + .. + a_nx_n$.

De esta forma, obtenemos la intersección de dos ideales finitamente generados, por lo que podemos calcular la solución recursivamente. Hasta obtener la matriz formada por los generadores de la solución. Que es la matriz que `(solveWithIntersection (Vec xs) int)` devuelve.

4.3. Anillos Coherentes y Fuertemente Discretos

Dentro de los anillos coherentes, podemos considerar los anillos coherentes fuertemente discretos. Restringiendo la clase de anillos a la clase de fuertemente discreto, nos permite una mayor facilidad a la hora de resolver sistemas. Puesto que, si un anillo es fuertemente discreto y coherente entonces podemos resolver ecuaciones del tipo $M\vec{X} = \vec{b}$. Lo vemos en el módulo `TAHCoherentSD`

```
module TAHCoherentSD
  ( solveGeneralEquation, propSolveGeneralEquation
  , solveGeneral, propSolveGeneral
  ) where

import Test.QuickCheck

import TAHIntegralDomain
import TAHIdeal
import TAHStronglyDiscrete
import TAHVector
import TAHMatrix
import TAHCoherent
```

Al estar en un anillo fuertemente discreto, la pertenencia al ideal es decidible de forma constructiva. Por lo que, si $x \in < x_1, \cdots, x_n >$ entonces existen unos coeficientes w_i de forma que x puede escribir como combinación lineal de los generadores del ideal. Entonces, podemos escribir x como $x = \sum_i w_i x_i$.

Proposición 4.3.1. *Si R es un dominio de integridad fuertemente discreto y coherente entonces es posible resolver sistemas lineales arbitrarios. Dado $M\vec{X} = \vec{b}$ es posible calcular $\vec{X}_0$ y L tal que $ML = \vec{0}$ y*

$$M\vec{X} = \vec{b} \Leftrightarrow \exists\, \vec{Y} \;/\; \vec{X} = L\vec{Y} + \vec{X}_0$$

Prueba 4.3.1. *Por coherencia, podemos calcular la matriz L del sistema $M\vec{X} = \vec{0}$ mediante la proposición (4.2.1). La solución particular $\vec{X}_0$ puede calcularse utilizando el siguiente método que utilizaremos para encontrar la solución de $\vec{X}$:*

El caso base es cuando M solo tiene una fila, la denotamos $\vec{m}$. Aquí es trivial ya que R es fuertemente discreto. Esto es, si $\vec{m} = (m_1, \cdots, m_n)$ y $\vec{b} = (b)$ entonces resolver $\vec{m}X = (b)$ es decidir si (b) pertenece al ideal $< m_1, \cdots, m_n >$ o no.
Si $b \in < m_1, \cdots, m_n >$ entonces se tiene que obtener los coeficientes w_i tales que $b = m_1 w_1 + \cdots + m_n w_n$. Por tanto, $(w_1, \cdots, w_n)$ es solución.

Mediante la función que denotaremos `solveGeneralEquation` obtendremos el primer paso para calcular la solución de un sistema del tipo $M\vec{X} = \vec{b}$, partiendo de que estamos en un anillo fuertemente discreto. Esta función recibe el vector v y la solución b. Aplicamos `solve` sobre dicho vector para encontrar la matriz L para verificar que se trata de un anillo coherente. Después con `member` se generará la lista de coeficientes de la combinación lineal. Finalmente se suman ambas.

```
solveGeneralEquation :: (Coherent a, StronglyDiscrete a) =>
                        Vector a -> a -> Maybe (Matrix a)
solveGeneralEquation v@(Vec xs) b =
  let sol = solve v  -- obtenemos la matriz L
  in case b `member` (Id xs) of
    Just as -> Just $ transpose
       (M (replicate (length (head (unMVec sol))) (Vec as)))
       `addM` sol
        -- Suma a L a los coeficientes de la comb. lineal de b
    Nothing -> Nothing


isSolutionB :: (CommutRing a, Eq a) => Vector a -> Matrix a -> a -> Bool
isSolutionB v sol b =
  all (==b) $ concat $ unMVec $ vectorToMatrix v `mulM` sol


-- | Comprueba que al multiplicar el vector v por la
--   matriz generada por solveGeneralEquation se obtiene el vector b
propSolveGeneralEquation :: (Coherent a, StronglyDiscrete a, Eq a)
                         => Vector a
                         -> a
                         -> Bool
propSolveGeneralEquation v b = case solveGeneralEquation v b of
  Just sol -> isSolutionB v sol b
  Nothing  -> True
```

La función `isSolutionB` es similar a `isSolution`. Ambas tienen el mismo objetivo, comprobar que la solución del sistema obtenida es correcta. Solo que una es para sistemas no homogéneos y la otra es para sistemas homogéneos, respectivamente.

Ahora vamos a resolver sistemas lineales generales de la forma $M\vec{X} = \vec{b}$.

```
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 verificamos
  -- que la nueva solución 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 of
         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"
```

Las dos funciones anteriores, `solveGeneralEquation` y `solveGeneral` también son similares. La diferencia es que `solveGeneralEquation` resuelve un sistema de la forma $\vec{m}\vec{X} = (b)$, es decir, para el caso en el que M es un vector y no una matriz. Mientras que `solveGeneral` nos permite calcular la solución de un sistema $M\vec{X} = \vec{b}$ donde M es una matriz. Ambas funciones están basadas en la proposición 4, solo que cada una es un caso de la prueba 4.

La función `solveGeneral` consiste en obtener el sistema de generadores de la solución. Para ello con `solveGeneralEquation` conseguimos un generador de la solución, a partir del cuál se comprueba que sea un generador para todas las ecuaciones del sistema.

Finalmente, con la siguiente propiedad comprobaremos que la solución es correcta. Primero tenemos que comprobar que las filas de M son de la misma longitud que $\vec{b}$. Después, multiplicamos la matriz M con la matriz solución y vemos si coincide componente a componente con el vector $\vec{b}$.

```
propSolveGeneral :: (Coherent a, StronglyDiscrete a, Eq a) =>
                    Matrix a -> Vector a -> Property
propSolveGeneral m b = length (unM m) == length (unVec b) ==>
  case solveGeneral m b of
   Just (l,x) -> all (==b) (unM (transpose (m `mulM` x))) &&
                 isSolution m l
```

```
Nothing -> True
```

Apéndice A

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.

A.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.
  - Seleccionamos 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.

A.2. Iniciar un Capítulo

El editor de Emacs podemos descargarlo de su página web, la instalación es muy sencilla y guiada. Del mismo modo, instalamos Haskell 8.0.2. Una vez tenemos ambos programas instalados, procedemos a iniciar un capítulo en Haskell literario.

- **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'}
  ```

Apéndice B

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

Apéndice C

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`

Bibliografía

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

[2] J. Alonso. Tipos y clases en Haskell. Technical report, Univ. de Sevilla, 2016.

[3] T. Coquand. Coherent Ring. Technical report, University of Gothenburg, 2010.

[4] M. Gago. Sistemas de ecuaciones lineales. Technical report, Univ. de Sevilla, 2015.

[5] J. González-Meneses. Tema 3: Anillos. Technical report, Univ. de Sevilla, 2015.

[6] A. Mörtberg. "Constructive Algebra in Functional Programming and Type Theory". Tesis de máster, University of Gothenburg, 2010.

[7] G. Hutton. "Programming in Haskell". Cambridge University Press, 2016.

[8] A. Serrano. "Beginning in Haskell". Utrecht University, 2014.

[9] "¡Aprende haskell por el bien de todos!". Technical report, aprendehaskell.es, 2018

[10] Haskell Language. haskell.org, 2018

Índice alfabético