, shadow , spanish]todonotes

Álgebra constructiva en Haskell

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

Autor

Agradecimientos

El presente Trabajo Fin de Grado se ha realizado en el Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla.

Supervisado por

Tutor

Abstract

Resumen en inglés

Índice general

Capítulo 1

Programación funcional con Haskell

En este capítulo se hace una breve introducción a la programación funcional en Haskell suficiente para entender su aplicación en los siguientes capítulos. Para una introducción más amplia se pueden consultar los apuntes de la asignatura de Informática de 1º del Grado en Matemáticas ([1]).

El contenido de este capítulo se encuentra en el módulo `PFH`

```
module PFH where
import Data.List
```

1.1. Introducción a Haskell

En esta sección se introducirán funciones básicas para la programación en Haskell. Como método didáctico, se empleará la definición de funciones ejemplos, así como la redefinición de funciones que Haskell ya tiene predefinidas, con el objetivo de que el lector aprenda "*a montar en bici, montando*".

A continuación se muestra la definición `(cuadrado x)` es el cuadrado de `x`. Por ejemplo, La definición es

```
-- |
-- >>> cuadrado 3
-- 9
-- >>> cuadrado 4
-- 16
cuadrado :: Int -> Int
cuadrado x = x * x
```

A continuación se muestra la definición `(cubo x)` es el cuadrado de `x`. Por ejemplo, La definición es

```
-- |
-- >>> cubo 3
-- 27
-- >>> cubo 2
-- 8
-- >>> cubo 4
-- 64
cubo :: Int -> Int
cubo x = x^3
```

S continuación se muestra la definición `(suma x y)` es la suma de `x` e `y`. Por ejemplo, La definición es

```
-- |
-- >>> suma 3 5
-- 8
-- >>> suma 4 2
-- 6
suma :: Int -> Int -> Int
suma x y = x + y
```

.

Capítulo 2

Como empezar con Emacs en Ubuntu

En este capítulo se hace una breve explicación de conceptos básicos para empezar a redactar un documento a LaTeX en Emacs y con Haskell a la vez, así como ir actualizando los archivos junto con la plataforma Github. Comenzaremos explicando como realizar la instalación de Ubuntu 16.04 en un PC con windows 10.

2.0.1. Instalar Ubuntu 16.04 junto a windows 10

Para realizar la instalación de Ubuntu junto a windows necesitaremos los siguientes programas:

+ Rufus-2.17

+ Ubuntu 16.04

También necesitaremos un pen drive para usarlo de instalador.

Paso 1:
Descargamos Ubuntu 16.04 y rufus-2.17 desde sus respectivas web (o enlaces dados anteriormente).
Necesitamos saber que tipo tiene nuestro disco duro, esto lo podemos ver haciendo click derecho sobre el icono de windows (abajo izquierda) y le damos a administrador de equipos ->administrador de discos, y nos aparecerá nuestro disco duro con todas sus subparticiones internas, en el general nos pondrá si es NTFS o MBR.

Paso 2:
Conectamos el pen al PC y abrimos el programa rufus, el propio programa reconocerá el pen, sino en la pestaña de dispositivo marcamos el pen.

En Tipo de partición si nuestro disco es NTFS marcamos GPT para UEFI, en caso contrario uno de los otros dos MBR.

En la parte de opciones de formateo marcamos (aunque deben de venir marcadas):

- Formateo rápido

- Crear disco de arranque con ->seleccionamos imagen ISO y con hacemos click en el icono de la derecha para adjuntar la imagen ISO de Ubuntu que hemos descargado anteriormente.

- Añadir etiquetas extendidas e iconos.

Y le damos a empezar.

Paso 3:
Dejamos el pen conectado al PC y reiniciamos el ordenador, al reiniciar justo antes de que cargue pulsamos F2 (o F1 según el PC) para acceder a la bios del PC y aqui nos vamos a la zona de arranque de cada sistema (esto cada bios es diferente) y tenemos que colocar el pen en la primera posición que en esta debe estar windows de esta forma iniciamos con el pen y comenzamos a instalar Ubuntu, seguimos los pasos solo tenemos que marcar España cuando nos lo pida y dar el espacio que queramos a Ubuntu con unos 40 GB sobra, el propio Ubuntu se encarga de hacer la partición del disco duro.

Paso 4:
Una vez instalado Ubuntu, nos vamos al icono naranjita que se llama software de Ubuntu y actualizamos.

Tras realizar todos estos pasos, cuando iniciemos el PC nos debe dar a elegir entre iniciar con Ubuntu o con Windows 10. Recomiendo buscar en youtube un video tuto-

rial de como instalar Ubuntu junto a windows 10.

2.0.2. Iniciar un Capítulo

Paso 1:
Abrimos el directorio desde Emacs con Ctrl+x+d y accedemos a la carpeta de texto para crear el archivo nuevo .tex sin espacios.

Paso 2:
Hacemos lo mismo pero en la carpeta código y guardamos el archivo con la abreviatura que hemos usado en el .tex, el archivo lo guardamos como .lhs para tener ahí el código necesario de Haskell.

Paso 3:
Al acabar el capitulo hay que actualizar el trabajo para que se quede guardado, para ello nos vamos a archivo que contiene todo el trabajo que en nuestro caso se llama 'TFG.tex' importante coger el de la extensión .tex, nos vamos a la zona donde incluimos los capitulos y usamos el comando de LaTeX con el nombre que le dimos en la carpeta de texto:

```
include{'nombre sin el .tex'}
```

2.0.3. Abreviaciones de Emacs:

La tecla ctrl se denominara C y la tecla alt M, son las teclas mas utilizadas, pues bien ahora explicamos los atajos más importantes y seguiremos la misma nomenclatura de la guía para las teclas:

ctrl es llamada C y alt M.

Para abrir o crear un archivo:
C + x + C + f

Para guardar un archivo:
C + x + C + s

Para guardar un archivo (guardar como):
C + x + C + w

Si abriste mas de un archivo puedes recorrerlos diferentes buffers con
C + x + ← o →

Emacs se divide y maneja en buffers y puedes ver varios buffers a la vez (los buffers son como una especie de ventanas).

Para tener 2 buffers horizontales:
C + x + 2

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

Para cambiar el puntero a otro buffer:
C + x + o

Para tener un solo buffer:
C + x + 1

Para cerrar un buffer:
C + x + k

Si por ejemplo nos equivocamos en un atajo podemos cancelarlo con:
C + g

Para cerrar emacs basta con:
C + x + C + C

Para suspenderlo:
C + z

Podemos quitar la suspensión por su id que encontraremos ejecutando el comando:

jobs

Y después ejecutando el siguiente comando con el id de emacs:

fg

Escribimos shell y damos enter.

2.0.4. Push and Pull de Github con Emacs

Vamos a mostrar como subir y actualizar los archivos en la web de Github desde la Consola (o Terminal), una vez configurado el pc de forma que guarde nuestro usuario y contraseña de Github. Lo primero que debemos hacer es abrir la Consola:

Ctrl+Alt+T

Escribimos los siguientes comandos en orden para subir los archivos:

cd 'directorio de la carpeta en la que se encuentran las subcarpetas de código y texto'
ejemplo: cd Escritorio/AlgebraConEnHaskell/

git add . (de esta forma seleccionamos todo)

git commit -m 'nombre del cambio que hemos hecho'

git push origin master

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

```
git pull origin master
```

El contenido de este capítulo se encuentra en el módulo `ICH`

```
module ICH where
import Data.List
```

.

Capítulo 3

Teoría de anillos en Haskell

En este capítulo se muestra cómo definir la teoría de anillos en Haskell. Los anillos se pueden definir de forma compacta en grupos y monoides, pero daremos unas series de definiciones que los definen de forma más rigurosa.

Comenzamos dando las primeras definiciones y propiedades básicas que tiene un anillo en el módulo `TAH`

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

import Data.List
import Test.QuickCheck
```

3.1. Teoría de anillos en Haskell

En esta sección, que corresponde con el fichero TAH.lhs, introducimos los conceptos básicos de la Teoría de Anillos. El objetivo es definir los conceptos mediante programación funcional y teoría de tipos.

En primer lugar, damos la definición teórica de anillos:

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

1. *Asociatividad de la suma:* $\forall\ a, b, c \in R.\ \ (a + b) + c = a + (b + c)$

2. *Existencia del elemento neutro para la suma:* $\exists\ 0 \in R.\ \forall\ a \in R.\ 0 + a = a + 0 = a$

3. *Existencia del inverso para la suma:* $\forall\ a \in R, \exists\ b \in R.\ a + b = b + a = 0$

4. *La suma es commutativa:* $\forall\ a, b \in R.\ a + b = b + a$

5. *Asociatividad de la multiplicación:* $\forall\ a, b, c \in R.\ \ (a * b) * c = a * (b * c)$

6. *Existencia del elemento neutro para la multiplicación:*

$$\exists\ 1 \in R.\ \ \forall\ a \in R.\ \ 1 * a = a * 1 = a$$

7. *Propiedad distributiva a la izquierda de la multiplicación sobre la suma:*

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

8. *Propiedad distributiva a la derecha de la multiplicación sobre la suma:*

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

Representaremos la noción de anillo en Haskell mediante una clase. Para ello, declaramos la clase Ring sobre un tipo a con las operaciones internas que denotaremos con los símbolos $< + >$ y $< ** >$ (nótese que de esta forma no coinciden con ninguna operación previamente definida en Haskell). Representamos el elemento neutro de la suma mediante la constante *zero* y el de la multiplicación mediante la constante *one*. Asímismo, mediante la operacion *neg* representamos el elemento inverso para la suma.

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

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

```
   one :: a
-- |1. Asociatividad de la suma.
propAddAssoc :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propAddAssoc a b c = ((a <+> b) <+> c == a <+> (b <+> c), "propAddAssoc")
-- |2. Existencia del elemento neutro para la suma.
propAddIdentity :: (Ring a, Eq a) => a -> (Bool,String)
propAddIdentity a = (a <+> zero == a && zero <+> a == a, "propAddIdentity")
-- |3. Existencia del inverso para la suma.
propAddInv :: (Ring a, Eq a) => a -> (Bool,String)
propAddInv a = (neg a <+> a == zero && a <+> neg a == zero, "propAddInv")
-- |4. La suma es commutativa.
propAddComm :: (Ring a, Eq a) => a -> a -> (Bool,String)
propAddComm x y = (x <+> y == y <+> x, "propAddComm")
-- |5. Asociatividad de la multiplicación.
propMulAssoc :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propMulAssoc a b c = ((a <**> b) <**> c == a <**> (b <**> c), "propMulAssoc")
-- |6. Existencia del elemento neutro para la multiplicación.
propMulIdentity :: (Ring a, Eq a) => a -> (Bool,String)
propMulIdentity a = (one <**> a == a && a <**> one == a, "propMulIdentity")
-- |7. Propiedad distributiva a la izquierda de la multiplicación sobre la suma.
propRightDist :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propRightDist a b c =
  ((a <+> b) <**> c == (a <**> c) <+> (b <**> c), "propRightDist")
-- |8. Propiedad distributiva a la derecha de la multiplicación sobre la suma.
propLeftDist :: (Ring a, Eq a) => a -> a -> a -> (Bool,String)
propLeftDist a b c =
 (a <**> (b <+> c) == (a <**> b) <+> (a <**> c), "propLeftDist")
```

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

```
-- | Test para ver que se satisfacen los axiomas de los anillos.
propRing :: (Ring a, Eq a) => a -> a -> a -> Property
propRing a b c = whenFail (print errorMsg) cond
  where
  (cond,errorMsg) =
    propAddAssoc a b c &&& propAddIdentity a  &&& propAddInv a        &&&
    propAddComm a b    &&& propMulAssoc a b c &&& propRightDist a b c &&&
    propLeftDist a b c &&& propMulIdentity a
  (False,x) &&& _         = (False,x)
  _         &&& (False,x) = (False,x)
  _         &&& _         = (True,"")
```

Veamos algunos ejemplos de anillos. Para ello, mediante instancias, especificamos las operaciones que dotan al conjunto de estructura de anillo. Por ejemplo, el anillo de los números enteros $\mathbb{Z}$ (en Haskell es el tipo *Integer*), con la suma y la multiplicación.

Ejemplo:

```
-- | El anillo de los enteros con la operaciones usuales:
instance Ring Integer where
     (<+>)  = (+)
     (<**>) = (*)
     neg    = negate
     zero   = 0
     one    = 1
```

Veamos ahora cómo definir nuevas operaciones en un anillo a partir de las propias del anillo. En particular, vamos a definir la diferencia, la potencia, etc. Estas operaciones se heredan a las instancias de la clase anillo y, por tanto, no habría que volver a definirlas para cada anillo particular.

En primer lugar, establecemos el orden de prioridad para los símbolos que vamos a utilizar para denotar las operaciones.

```
infixl 8 <^>
infixl 6 <->
infixl 4 ~~
infixl 7 <**
```

```
-- | Diferencia.
(<->) :: Ring a => a -> a -> a
a <-> b = a <+> neg b
-- | Suma de una lista de elementos.
sumRing :: Ring a => [a] -> a
sumRing = foldr (<+>) zero
-- | Producto de una lista de elementos.
productRing :: Ring a => [a] -> a
productRing = foldr (<**>) one
-- | Potencia.
(<^>) :: Ring a => a -> Integer -> a
x <^> 0 = one
x <^> y | y < 0     = error "<^>: La entrada debe ser positiva."
        | otherwise = x <**> x <^> (y-1)
-- | Relación de semi-igualdad: dos elementos son semi-iguales si son
--   iguales salvo el signo.
(~~) :: (Ring a, Eq a) => a -> a -> Bool
x ~~ y = x == y || neg x == y || x == neg y || neg x == neg y
```

Finalmente definimos la multiplicación de un entero por la derecha, la multiplicación de un entero por la izquierda se tiene debido a que la operación $< + >$ es

commutativa.

```
-- |Multiplicación de un entero por la derecha.
(<**) :: Ring a => a -> Integer -> a
_ <** 0 = zero
x <** n | n > 0     = x <+> x <** (n-1)
        | otherwise = neg (x <** abs n) -- error "<**: Entrada Negativa."
```

Para describir los anillos commutativos necesitamos un nuevo módulo `TAHCommutative`

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

import Test.QuickCheck
import TAH
```

En este módulo introducimos el concepto de anillo conmutativo, que visto desde el punto de vista de la programación funcional, es una subclase de la clase *Ring*. Solo necesitaremos una función para definirlo, damos primero su definición teórica.

Definición 2. *Un anillo conmutativo es un anillo* $(R, +, *)$ *en el que la operación de multiplicación* $*$ *es conmutativa, es decir,* $\forall\ a, b \in R.\ a * b = b * a$

```
class Ring a => CommutRing a
propMulComm :: (CommutRing a, Eq a) => a -> a -> Bool
propMulComm a b = a <**> b == b <**> a
```

Para saber si un anillo es commutativo se necesita una función que verifique la propiedad:

```
-- | Test que comprueba si un anillo es commutativo.
propCommutRing :: (CommutRing a, Eq a) => a -> a -> a -> Property
propCommutRing a b c = if propMulComm a b
                               then propRing a b c
                               else whenFail (print "propMulComm") False
```

3.2. Dominio de integridad y Cuerpos.

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

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

import Test.QuickCheck
import TAHCommutative
```

Definición 3. *Dado un anillo A, un elemento $a \in A$ se dice que es un divisor de cero si existe $b \in A - \{0\}$ tal que $a * b = 0$. Un anillo A se dice dominio de integridad, si el único divisor de cero es 0. Es decir, $\forall\ a, b \in R.\ a * b = 0 \Rightarrow\ a = 0$ or $b = 0$*

```
-- | Definición de dominios integrales.
class CommutRing a => IntegralDomain a
-- | Un dominio integral es un anillo que
propZeroDivisors :: (IntegralDomain a, Eq a) => a -> a -> Bool
propZeroDivisors a b = if a <**> b == zero then
                            a == zero || b == zero else True
```

Para determinar si un anillo es un dominio de integridad usaremos la siguiente función:

```
propIntegralDomain :: (IntegralDomain a, Eq a) => a -> a -> a -> Property
propIntegralDomain a b c = if propZeroDivisors a b
                           then propCommutRing a b c
                           else whenFail (print "propZeroDivisors") False
```

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

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

import Test.QuickCheck
import TAHIntegralDomain
```

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

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

Para saber si un anillo conmutativo es un cuerpo usaremos la función:

```
propField :: (Field a, Eq a) => a -> a -> a -> Property
propField a b c = if propMulInv a
                  then propIntegralDomain a b c
                  else whenFail (print "propMulInv") False
```

En un cuerpo se puede definir la división. Para poder dar dicha definición establecemos el orden de prioridad para el símbolo de la división.

```
infixl 7 </>

-- | División
(</>) :: Field a => a -> a -> a
x </> y = x <**> inv y
```

3.3. Ideales

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

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

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

```
  ) where

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

import TAHCommutative
```

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

Definición 6. *Sea R un anillo, y E un subconjunto de R. Se define el ideal generado por E, y se denota $< E >$, como la intersección de todos los ideales que contienen a E (que es una familia no vacía puesto que R es un ideal que contiene a E).*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
                          , [replicate (length xs) zero]
                          , [replicate (length ys) zero])
```

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

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

import TAHCommutative
import TAHIdeal
```

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

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

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

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

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

Capítulo 4

Anillos coherentes

Todos los anillos en esta sección son dominios integrales. Uno de los principales objetivos de las siguientes secciones serán para demostrar que los diferentes anillos son coherentes. Eso significa que es posible resolver sistemas de ecuaciones en ello. Lo vemos implementado en `TAHCoherent`

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

import Test.QuickCheck

import Algebra.Structures.IntegralDomain
import Algebra.Structures.StronglyDiscrete
import Algebra.Matrix
import Algebra.Ideal
```

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

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

De esta forma es posible calcular un conjunto de generadores para soluciones de ecuaciones en un anillo coherente. En otras palabras, el módulo de lasolución para $MX = 0$ esta generado finitamente.

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

```
-- | Test that the second matrix is a solution to the first.
isSolution :: (CommutativeRing 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)

-- | Solves a system of equations.
solveMxN :: (Coherent a, Eq a) => Matrix a -> Matrix a
solveMxN (M (l:ls)) = solveMxN' (solve l) ls
  where
  -- Inductively solve all subsystems. If the computed solution is in fact a
  -- solution to the next set of equations then don't do anything.
  -- This solves the problems with having many identical rows in the system,
  -- like [[1,1],[1,1]].
  solveMxN' :: (Coherent a, Eq a) => Matrix a -> [Vector a] -> Matrix a
  solveMxN' m []      = m
  solveMxN' m1 (x:xs) = if isSolution (vectorToMatrix x) m1
                        then solveMxN' m1 xs
                        else solveMxN' (m1 `mulM` m2) xs
    where m2 = solve (matrixToVector (mulM (vectorToMatrix x) m1))


-- | Test that the solution of an MxN system is in fact a solution of the system
propSolveMxN :: (Coherent a, Eq a) => Matrix a -> Bool
propSolveMxN m = isSolution m (solveMxN m)


--------------------------------------------------------------------------------
-- | Intersection computable -> Coherence.
--
-- Proof that if there is an algorithm to compute a f.g. set of generators for
-- the intersection of two f.g. ideals then the ring is coherent.
--
-- Takes the vector to solve, \[x1,...,xn\], and a function (int) that computes
-- the intersection of two ideals.
--
-- If \[ x_1, ..., x_n \] \`int\` \[ y_1, ..., y_m \] = \[ z_1, ..., z_l \]
--
-- then int should give witnesses us and vs such that:
--
--     z_k = n_k1 * x_1 + ... + u_kn * x_n = u_k1 * y_1 + ... + n_km * y_m
--
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: Can't solve an empty system"
  solveInt [x]    = [[zero]] -- Base case, could be [x,y] also...
                             -- That wouldn't give the trivial solution...
  solveInt [x,y]  | x == zero || y == zero = [[zero,zero]]
                  | otherwise =
    let (Id ts,us,vs) = (Id [x]) `int` (Id [neg y])
    in [ u ++ v | (u,v) <- zip us vs ]
  solveInt (x:xs)
    | x == zero             = (one : replicate (length xs) zero) : (map (zero:) $ sol
    | isSameIdeal int as bs = s ++ m'
    | otherwise             = error "solveInt: This does not compute the intersection
      where
      as            = Id [x]
      bs            = Id (map neg xs)

      -- Compute the intersection of <x1> and <-x2,...,-xn>
      (Id ts,us,vs) = as `int` bs
      s             = [ u ++ v | (u,v) <- zip us vs ]

      -- Solve <0,x2,...,xn> recursively
      m             = solveInt xs
      m'            = map (zero:) m


-------------------------------------------------------------------------------
-- | Strongly discrete coherent rings.
--
-- If the ring is strongly discrete and coherent then we can solve arbitrary
-- equations of the type AX=b.
--
solveGeneralEquation :: (Coherent a, StronglyDiscrete a) => Vector a -> a -> Maybe (M
solveGeneralEquation v@(Vec xs) b =
  let sol = solve v
  in case b `member` (Id xs) of
    Just as -> Just $ transpose (M (replicate (length (head (unMVec sol))) (Vec as)))
                      `addM` sol
    Nothing -> Nothing


propSolveGeneralEquation :: (Coherent a, StronglyDiscrete a, Eq a)
                         => Vector a
                         -> a
                         -> Bool
```

```
propSolveGeneralEquation v b = case solveGeneralEquation v b of
  Just sol -> all (==b) $ concat $ unMVec $ vectorToMatrix v `mulM` sol
  Nothing  -> True


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


-- | Solves general linear systems of the kind AX = B.
--
-- A is given as a matrix and B is given as a row vector (it should be column
-- vector).
--
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
  -- Compute a new solution inductively and check that the new solution
  -- satisfies all the previous equations.
  solveGeneral' Nothing _ _ _              = Nothing
  solveGeneral' (Just m) [] [] old         = Just m
  solveGeneral' (Just m) (l:ls) (a:as) old =
    if isSolutionB l m a
       then solveGeneral' (Just m) ls as old
       else case solveGeneralEquation (matrixToVector (vectorToMatrix l `mulM` m)) a
         Just m' -> let m'' = m `mulM` m'
                    in if all (\(x,y) -> isSolutionB x m'' y) old
                          then solveGeneral' (Just m'') ls as ((l,a):old)
                          else Nothing
         Nothing -> Nothing
  solveGeneral' _ _ _ _ = error "solveGeneral: Bad input"

-- It would be great to only generate solvable systems...
-- propSolveGeneral :: (Coherent a, StronglyDiscrete a, Eq a) => Matrix a -> Vector a
propSolveGeneral m b = length (unM m) == length (unVec b) ==> case solveGeneral m b o
  Just (l,x) -> all (==b) (unM (transpose (m `mulM` x))) &&
                isSolution m l
  Nothing -> True
```

Bibliografía

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

Índice alfabético