Divisibilidad y congruencia

Taller de Álgebra I

Segundo cuatrimestre de 2016

Algoritmo de la división

Teorema

Dados a, $d \in \mathbb{Z}$, $d \neq 0$, existen únicos $q, r \in \mathbb{Z}$ tales que

- $a = qd + r$,
- $0 \leq r < |d|$.

Idea de la demostración: (caso $a \geq 0$, $d > 0$). Por inducción en a.

- Si $0 \leq a < d$, tomamos $q = 0$, $r = a$.
- Si no, dividimos $a - d$ por d. Eso da un cociente q' y un resto r'. Tomamos $r = r'$, $q = q' + 1$.

Números primos

Definición

Un entero $p > 1$ es **primo** si ningún natural k tal que $1 < k < p$ divide a p.

¿Cómo determinamos en Haskell si un número entero mayor que cero es primo? Una posibilidad sería buscar sus divisores.

$$\mathit{divisores}(n) = \{k \in \mathbb{Z} \mid 1 \leq k \leq n \text{ y } k|n\}$$

¿Cómo lo hacemos de forma recursiva? **¿Sirve hacer recursión sobre n?**

Idea: definir una lista "parcial" de divisores:

$$\mathit{divParcial}(n, m) = \{k \in \mathbb{Z} \mid 1 \leq k \leq m \text{ y } k|n\}$$

En este caso, haríamos recursión sobre m.

Algoritmo de Euclides

El **algoritmo de Euclides** calcula el máximo común divisor entre dos números $a, b \in \mathbb{Z}$.

Se basa en que si a, $b \in \mathbb{Z}$ y $k \in \mathbb{Z}$ es un número cualquiera, entonces

$$(a \ : \ b) = (a + kb \ : \ b)$$

Si q y r son el cociente y el resto de la división de a por b, tenemos $a = qb + r$, entonces $a - qb = r$. Por lo tanto,

$$(a \ : \ b) = (a - qb \ : \ b) = (r \ : \ b) = (b \ : \ r)$$

Por ejemplo, para calcular $(30 \ : \ 48)$:

1. $(30 \ : \ 48)$ — Dividimos 30 por 48, $q = 0$, $r = 30$
2. $= (48 \ : \ 30)$ — Dividimos 48 por 30, $q = 1$, $r = 18$
3. $= (30 \ : \ 18)$ — $q = 1$, $r = 12$
4. $= (18 \ : \ 12)$ — $q = 1$, $r = 6$
5. $= (12 \ : \ 6)$ — $q = 2$, $r = 0$
6. $= (6 \ : \ 0)$
7. $= 6$

Divisibilidad: Ejercicios

Ejercicios

1. Implementar la función
   `division :: Integer -> Integer -> (Integer, Integer)`
   `division a d` debe funcionar para `a` $\geq$ `0`, `d` $>$ `0`, y no se pueden usar `div`, `mod` ni `(/)`.

2. (a) Implementar la función
   `divParcial :: Integer -> Integer -> [Integer]`
   `divParcial n m` debe funcionar bien siempre que `0` $<$ `m` $\leq$ `n`.

   (b) Utilizando `divParcial`, programar
   `divisores :: Integer -> [Integer]`

   (c) Utilizando `divisores`, programar
   `esPrimo :: Integer -> Bool`

3. (a) Programar la función
   `mcd :: Integer -> Integer -> Integer`
   que utilice el algoritmo de Euclides calcule el máximo común divisor entre dos números.
   `mcd a b` debe funcionar siempre que `a` $>$ `0`, `b` $\geq$ `0`.

   (b) Programar la función
   `euclides :: Integer -> Integer -> (Integer, Integer)`
   que utilice el algoritmo de Euclides extendido para obtener dos valores (s, t) tales que $(a : b) = sa + tb$.

Clases de congruencia

Llamamos **clase de congruencia** al conjunto de todos los enteros congruentes a un cierto número, módulo otro.

Ejemplos:

- $\{a \in \mathbb{Z} \mid a \equiv 1 \pmod 3\} = \{\ldots, -2, 1, 4, 7, 10, \ldots\}$,
- $\{a \in \mathbb{Z} \mid a \equiv 0 \pmod 9\} = \{\ldots, -18, -9, 0, 9, 18, 27, \ldots\}$.

Cada clase de congruencia es un *conjunto infinito* de enteros tales que entre dos elementos consecutivos del conjunto hay siempre la misma diferencia.

Consideraremos también una clase de congruencia válida al conjunto vacío ($\varnothing$).

En Haskell, vamos a modelar las clases de congruencia con el siguiente tipo de datos:

```
data ClaseCongr = Vacio | CongruentesA Integer Integer (deriving
    Show)
```

Así, por ejemplo,

- $\{a \in \mathbb{Z} \mid a \equiv 1 \pmod 3\}$ es `CongruentesA 1 3`,
- $\{a \in \mathbb{Z} \mid a \equiv 0 \pmod 9\}$ es `CongruentesA 0 9`.

Clases de congruencia: Ejercicios

Ejercicios

A partir del tipo de datos

```
data ClaseCongr = Vacio | CongruentesA Integer Integer (deriving
    Show)
```

programar las siguientes funciones:

1. `multiplo :: Integer -> Integer -> Bool`, que determine si el primero de sus argumentos es múltiplo del segundo. (¡Cuidado con el 0!).

2. `congruentes :: Integer -> Integer -> Integer -> Bool`, que verifique si sus dos primeros argumentos son congruentes módulo el tercero. Asumir que el tercer argumento es distinto de cero.

3. `pertenece :: Integer -> ClaseCongr -> Bool`, que determine si un entero forma parte de una clase de congruencia. Por ejemplo:
   `pertenece 13 Vacio` $\rightsquigarrow$ `False`
   `pertenece 13 (CongruentesA 5 4)` $\rightsquigarrow$ `True`

4. `incluido :: ClaseCongr -> ClaseCongr -> Bool`, que dadas dos clases de congruencia P_1 y P_2, determine si $P_1 \subseteq P_2$. Por ejemplo:
   `incluido (CongruentesA 4 6) (CongruentesA 10 3)` $\rightsquigarrow$ `True`

El regreso de las clases de tipos

En las últimas clases, vimos como hacer que nuestros tipos formen parte de algunas clases de tipos, como `Show`, `Eq` u `Ord`. Para esto, usamos la palabra clave `deriving`; así, Haskell define de forma automática las funciones necesarias.

¿Cómo hacemos para definir **nosotros mismos** estas funciones? Usamos la palabra clave `instance`.

Por ejemplo, en lugar de:

```
data ClaseCongr = Vacio | CongruentesA Integer Integer (deriving
    Show)
```

podemos hacer lo siguiente:

```
data ClaseCongr = Vacio | CongruentesA Integer Integer
instance Show ClaseCongr where
    show Vacio = "Progresion vacia"
    show (CongruentesA x d) = "Progresion no vacia"
```

Para pensar entre todos

¿En qué otros casos resulta interesante definir nuestras propias funciones para `Show`, `Eq` u `Ord`, en lugar de utilizar las que Haskell infiere por defecto?

Clases de tipos: Ejercicios

Ejercicios

1. Implementar la función `show` de la clase `ClaseCongr`, para que las clases de congruencia se muestren de esta forma:

```
Prelude> CongruentesA 3 8
{a en Z | a = 3 (mod 8)}
```

```
Prelude> Vacio
{}
```

2. Programar `iguales :: ClaseCongr -> ClaseCongr -> Bool`, que determina si dos clases de congruencia tienen los mismos elementos. Por ejemplo, `iguales (CongruentesA 22 5) (CongruentesA 2 5)` $\rightsquigarrow$ `True`.
3. Hacer que `ClaseCongr` sea instancia de `Eq`, utilizando la igualdad programada anteriormente.

Recordar:

- Para que un tipo `t` sea instancia de `Show`, hay que definir una función `show :: t -> String`.
- Para que un tipo `t` sea instancia de `Eq`, hay que definir una función `(==) :: t -> t -> Bool`.

Ejercicios

1. Si P_1 y P_2 son clases de congruencia, definimos su suma como

$$P_1 + P_2 = \{(a + b) \mid a \in P_1,\ b \in P_2\}$$

   Verificar que la suma de dos clases de congruencia es también una clase de congruencia, y programar una función `suma :: ClaseCongr -> ClaseCongr -> ClaseCongr` que calcule esta suma.

   Por ejemplo, `suma (CongruentesA 3 6) (CongruentesA 2 4)` $\rightsquigarrow$ `CongruentesA 5 2`.

2. Verificar que la intersección entre dos clases de congruencia ($P_1 \cap P_2$) también es una clase de congruencia, e implementar `interseccion :: ClaseCongr -> ClaseCongr -> ClaseCongr`, que calcule esta intersección.

3. Implementar `tieneSolucion :: Integer -> ClaseCongr -> Bool` que diga si una ecuación de congruencia tiene solución. Explícitamente, `tieneSolucion t (CongruentesA a m)` $\rightsquigarrow$ `True` si y solo si la ecuación $tx \equiv a \pmod{m}$ tiene solución.

Divisibilidad: Bonus track

1. Adaptar `division` para $a < 0$ o $d < 0$. Observación: las funciones `div` y `mod` de Haskell no coinciden con el algoritmo de división cuando $d < 0$. Ver también `quot` y `rem`.

2. (a) Implementar `noTieneDivisoresHasta :: Integer -> Integer -> Bool`. `noTieneDivisoresHasta m n` da `True` sii ningún número entre 2 y `m` divide a `n`.

   (b) Utilizando `noTieneDivisoresHasta`, programar `esPrimo' :: Integer -> Bool`.

3. (a) Escribir una función `primosHasta :: Integer -> [Integer]` que dado $n > 0$ devuelva la lista de todos los primos positivos menores o iguales que n.

   (b) Escribir una función `hayPrimosEntre :: Integer -> Integer -> Bool` tal que `hayPrimosEntre a b` es `True` si y solo si hay algún primo p con $a \leq p \leq b$.

   (c) Escribir una función `bertrand :: Integer -> Bool` que dado $n \geq 1$ diga si hay algún primo entre n y $2n$. No vale usar el postulado de Bertrand[1].

4. **Conjetura** (Christian Goldbach, 1742): Todo número par mayor que 2 puede escribirse como suma de dos números primos.

   (a) Dado un número natural n, determinar si puede escribirse como suma de dos números primos.

   (b) En función de este ejercicio, ¿cómo sería un programa en Haskell para testear la conjetura de Goldbach hasta un cierto punto?

[1] El postulado de Bertrand dice que esta función vale siempre `True`