Listas como conjuntos
Lógica Computacional 2017-2

Lourdes del Carmen González Huesca
Roberto Monroy Argumedo
Fernando A. Galicia Mendoza

Facultad de ciencias,UNAM

Miércoles, 15 de febrero del 2016

El problema

Supongamos que queremos una lista que contenga los primeros 15 números naturales, una forma para lograr esto es declarando la siguiente expresión:

```
l1 = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Esto resulta tedioso e ineficiente a la hora de programar. Para esto `Haskell` provee de sintaxis para este tipo de tareas, tal sintaxis se llama *listas por comprensión*.

Predicados

Primero recordemos que la expresión `if` está constituida de la siguiente forma:

Predicados

Primero recordemos que la expresión `if` está constituida de la siguiente forma:

```
if guardia_booleana then exprTrue else exprFalse
```

la guardia booleana son expresiones que representan un predicado, es decir, dada `p` una función que reciba un tipo `a` y devuelva un tipo `Bool`, nos hacen recordar un predicado.

Predicados

Primero recordemos que la expresión `if` está constituida de la siguiente forma:

```
if guardia_booleana then exprTrue else exprFalse
```

la guardia booleana son expresiones que representan un predicado, es decir, dada `p` una función que reciba un tipo `a` y devuelva un tipo `Bool`, nos hacen recordar un predicado.
Ya que, un predicado en matemáticas es una relación que resulta cierta en cierto modelo, en nuestro caso el predicado es cierto en el modelo representado por el tipo `a`.

Sintaxis

Ahora recordemos que es usual en matemáticas definir conjuntos que cumplan ciertas propiedades, por ejemplo:

$$\{x | x \in \mathbb{N} \wedge x \equiv 0 \ (mod\ 2)\}$$

Esto se traduce fácilmente a `Haskell`, ya que, la sintaxis de las listas por comprensión es la siguiente:

```
[e | q1,...,qn]
```

donde `n`≥ 1, `qi` es un predicado para $1 \leq$`i`$\leq$`n` y `e` representa las expresiones que van a estar almacenadas en la lista.

Sintaxis

Ahora recordemos que es usual en matemáticas definir conjuntos que cumplan ciertas propiedades, por ejemplo:

$$\{x | x \in \mathbb{N} \wedge x \equiv 0 \ (mod\ 2)\}$$

Esto se traduce fácilmente a `Haskell`, ya que, la sintaxis de las listas por comprensión es la siguiente:

```
[e | q1,...,qn]
```

donde `n`≥ 1, `qi` es un predicado para $1 \leq$`i`$\leq$`n` y `e` representa las expresiones que van a estar almacenadas en la lista.

```
[n | n ← [0..], par n]
```

Listas infinitas y acotadas

Se utilizó la expresión `[0..]`, esto indica a `Haskell` que es una lista infinita. Esto lo logra gracias a su mecanismo de evaluación perezosa. Ahora si se requiere definir una lista como la primera, la expresión que realiza esto es:

Listas infinitas y acotadas

Se utilizó la expresión `[0..]`, esto indica a `Haskell` que es una lista infinita. Esto lo logra gracias a su mecanismo de evaluación perezosa. Ahora si se requiere definir una lista como la primera, la expresión que realiza esto es:

```
l1 [1..15]
```

Es decir, acotamos una lista por ambos lados. Para lograr esto lo <<único>> que solicita `Haskell` es que el tipo encapsulado por las listas debe tener instanciada la clase `Enum`, es decir, que los elementos del tipo puedan ser secuencialmente ordenados.

Listas vs conjuntos

Sabemos que una lista cumple dos propiedades que los conjuntos no cumplen:

Listas vs conjuntos

Sabemos que una lista cumple dos propiedades que los conjuntos no cumplen:ser ordendadas y repetición de elementos.

Listas vs conjuntos

Sabemos que una lista cumple dos propiedades que los conjuntos no cumplen:ser ordendadas y repetición de elementos.
Chisme: Las estructuras de datos que cumplen tener repetición de elementos pero no tener orden, son llamadas multiconjuntos o bolsas.

Listas vs conjuntos

Sabemos que una lista cumple dos propiedades que los conjuntos no cumplen:ser ordendadas y repetición de elementos.
Chisme: Las estructuras de datos que cumplen tener repetición de elementos pero no tener orden, son llamadas multiconjuntos o bolsas.
Sabiendo esto la mejor forma de representar un conjunto en `Haskell` es una lista que no repita elementos, entonces para verificar que una lista es un conjunto, requerimos de una función que dada una lista verifique que no tenga elementos repetidos.

Una recursión mas eficiente

Observamos que en la función `elimRep`, utilizamos el operador de concatenación para no perder el orden original de la lista, sin embargo, esto resulta ineficiente por la definición de la concatenación.

Una recursión mas eficiente

Observamos que en la función `elimRep`, utilizamos el operador de concatenación para no perder el orden original de la lista, sin embargo, esto resulta ineficiente por la definición de la concatenación.Una forma de mejorar la función es hacer lo siguiente:

```
elimRep :: Eq a ⇒ [a] → [a]
elimRep = rev.eR_aux []
```

Donde `.` es el operador de composición de funciones, es decir, si `f x = g (h x)`, entonces se puede definir como `f = g.h`.
Otra forma de definirlo es con el operador sin paréntesis:

```
elimRep :: Eq a ⇒ [a] → [a]
elimRep x = rev $ eR_aux [] x
```

Una recursión mas eficiente

Observamos que en la función `elimRep`, utilizamos el operador de concatenación para no perder el orden original de la lista, sin embargo, esto resulta ineficiente por la definición de la concatenación.Una forma de mejorar la función es hacer lo siguiente:

```
elimRep :: Eq a ⇒ [a] → [a]
elimRep = rev.eR_aux []
```

Donde `.` es el operador de composición de funciones, es decir, si `f x = g (h x)`, entonces se puede definir como `f = g.h`.
Otra forma de definirlo es con el operador sin paréntesis:

```
elimRep :: Eq a ⇒ [a] → [a]
elimRep x = rev $ eR_aux [] x
```

Chisme: Con expresión lambda se puede definir:

```
elimRep :: Eq a ⇒ [a] → [a]
elimRep = λ x → rev $ eR_aux [] x
```

Operaciones sobre conjuntos

Ya que tenemos una forma de verificar si una lista está implementando de forma correcta un conjunto, lo mas natural es definir ciertas operaciones sobre conjuntos.
Para esto pasémonos a `Haskell`.