Universidad Nacional Autónoma de México

Facultad de Ciencias

Manufactura de tipos de datos mediante multiconjuntos

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LIC. EN CIENCIAS DE LA COMPUTACIÓN
PRESENTA:

MARTHA DANIELA ABIGAIL LAURO AGUILAR

DIRECTOR DE TESIS:
DR. FAVIO EZEQUIEL MIRANDA PEREA

2015

Ciudad Universitaria, D. F.

Agradecimiento

El desarrollo del presente trabajo de tesis tuvo lugar en el marco del proyecto

"Formalismos lógico - categóricos para la programación funcional"
(PAPIIT, IN117711)

otorgado por la Dirección General de Asuntos del Personal Académico
de la Universidad Nacional Autónoma de México.

Índice general

Introducción

Las estructuras de datos han tomado gran importancia en el diseño e implementación de algoritmos, por esta razón, es fundamental contar con estructuras de datos adecuadas para cada propósito particular.

Las estructuras de datos de propósito general, como listas o árboles, no imponen condiciones de tamaño o forma. Sin embargo, muchas estructuras de datos utilizadas con propósitos particulares necesitan de ciertas restricciones, ya sea en su tamaño (número de elementos que es posible almacenar) o en su estructura (forma de sus componentes). Algunas de estas restricciones se muestran en la siguiente tabla:

Estructura de Datos	**Restricción de tamaño**	**Restricción estructural**
Matrices cuadradas	n^2	El número de renglones y de columnas son el mismo.
Matrices de Toeplitz	n^2	Si M = (a_{ij}), entonces para 1¡i, j $\leq$ n, $a_{ij} = a_{i-1j-1}$
Árboles binarios perfectos	$2^{n+1} - 1$	Los subárboles deben ser perfectos
Árboles de Braun	n	El subárbol izquierdo tiene a lo más un nodo más que el subárbol derecho
Árboles híbridos	n	Son árboles binarios perfectos donde para cada nivel, todos o ninguno de sus nodos están etiquetados y los nodos del último nivel están etiquetados.

La manera usual de implementar esta clase de estructuras consiste en utilizar aquellas de propósito general, pero implementando las funciones necesarias para manejar la estructura pensando únicamente en sus instancias válidas. Por ejemplo, uno implementa matrices cuadradas en programación funcional, utilizando una lista de listas y pensando que las funciones requeridas, como la suma y multiplicación de matrices, solamente recibirán como entrada matrices válidas, es decir, una lista de listas de tamaño n en donde todos sus elementos también son de tamaño n. Si se sigue está metodología se debe verificar en todo momento que las entradas son válidas, lo cual se hace mediante una función que comprueba que las restricciones de la estructura se cumplan.

En contraste, en este trabajo se describe a través de la programación funcional pura, un proceso de diseño de estructuras de datos que satisfacen de manera estática las restricciones de forma o de tamaño, basado especialmente en el artículo *Manufacturing datatypes* de Ralf Hinze [3] . Por ejemplo, el tipo de matrices cuadradas obtenido bajo este procedimiento, únicamente permitirá definir listas de listas que corresponden a matrices cuadradas.

Si bien es de gran importancia utilizar está clase de estructuras, es decir, definir funciones sobres ellas, el propósito de nuestro trabajo consiste únicamente en definir y justificar un proceso matemático de manufactura de tipos de datos, que corresponden a estructuras de datos como las recién mencionadas. Para hacer ésto nos servimos de los conceptos matemáticos de multiconjunto y funtor, a partir de los cuales obtendremos la definición de un tipo de dato en el lenguaje de programación Haskell, el cual cumplirá los criterios de restricción dados.

El contenido del trabajo es el siguiente:

En el capítulo 1 mostramos los conceptos generales sobre la programación funcional y el lenguaje de programación Haskell. En particular nos interesa el mecanismo de definición de tipos de datos en este lenguaje.

En el capítulo 2 presentamos la definición formal de multiconjunto, incluyendo algunas de sus operaciones y propiedades. Adicionalmente estudiamos los multiconjuntos de números naturales, que son los que necesitamos para nuestro propósito principal. El concepto de funtor en Haskell así como algunas propiedades de interés se describen en el capítulo 3, siendo el resultado principal de este capítulo la correspondencia entre multiconjuntos y

funtores, base de nuestro proceso de manufactura.

Por último, el capítulo 4 lo dedicamos a explicar de forma concreta el proceso de manufactura de tipos de datos y nos servimos del análisis de algunos casos para ejemplificarlo, tales como: matrices de Toeplitz, matrices cuadradas, árboles de Braun y árboles híbridos.

Capítulo 1

Preliminares

1.1. Programación Funcional y Haskell

Este trabajo se desarrolla dentro del *paradigma de Programación Funcional*, donde un programa es simplemente una sucesión de definiciones de funciones y el método básico de cómputo es la aplicación de entre ellas.

Este tipo de programación requiere que las funciones sean entidades de primera clase, es decir, que puedan recibir como argumentos de entrada otras funciones o devolver como resultado otra función. Además, por ser individuos de primera clase, también es posible definirlas y manipularlas desde el cuerpo de otras funciones. Este enfoque produce programas considerablemente más simples que su contraparte en la programación imperativa u orientada a objetos, y permite nuevas y poderosas formas para estructurar programas y razonar acerca de sus propiedades.

Algunas características importantes de los lenguajes de programación funcional son:

- Un programa es declarativo, es decir, en la solución de un problema no importa el cómo sino el qué.
- La orientación es hacia la evaluación, es decir, el mecanismo básico de cómputo es la evaluación de funciones.
- El control se basa en recursión.

Por ejemplo, si se desea calcular las potencias de un número n, una función recursiva que hace el trabajo es la siguiente:

```
pot n 0 = 1
pot n (exp + 1) = n * (pot n exp)
```

y resulta ser que esta definición matemática corresponde exactamente a un programa funcional.

Los lenguajes de programación funcional se pueden agrupar en *puros* e *impuros*. Los puros se caracterizan por no tener efectos colaterales como el enunciado de asignación, lo cual implica que una función en el lenguaje es también una función en el sentido matemático; en contraste un lenguaje impuro sí posee efectos colaterales, los cuales pueden causar que una función en el lenguaje deje de ser una función matemática.

Dentro de los lenguajes de programación funcional puros se encuentran: Miranda, Gofer, Haskell, mientras que en los impuros están: Lisp, Scheme, ML. En particular, para el propósito de este trabajo se utilizará el lenguaje Haskell.

Haskell es un lenguaje de programación funcional puro, perezoso, polimórfico y estáticamente tipado, que fue definido como un intento de crear un estándar para el paradigma de programación funcional, su primera versión aparece en 1990. Su nombre se debe a Haskell Brooks Curry (1900-1982), lógico estadounidense cuyo trabajo acerca de la lógica combinatoria y el cálculo lambda resulta ser la piedra ángular sobre la que descansa la programación funcional.

En un lenguaje perezoso se realiza el cálculo de una expresión hasta que el valor de ésta sea realmente necesario. Por ejemplo, una función que duplique cada elemento de una lista, está definida de la siguiente forma:

```
duplicaElem [] = []
duplicaElem (x:xs) = 2*x : duplicaElem xs
```

De esta manera, si se desea obtener el primer elemento de la lista resultante de aplicar `duplicaElem` a [2,3,4], utilizamos la función `head` definida en el preludio de Haskell, y así el desarrollo de la evaluación queda como sigue:

```
head (duplicaElem [2,3,4]) = head ((2*2): duplicaElem [3,4])
                           = (2*2)
                           = 4
```

Así, es posible observar que para obtener el resultado final, solo se realiza un recorrido por la lista inicial para manipular y transformar sus elementos, y las operación de multiplicación requerida para obtener el resultado final se realiza únicamente al primer elemento y hasta que sea necesario. Esta estrategia de evaluación permite en particular definir y manipular estructuras infinitas.

Cuando es compilado un programa en HASKELL, el compilador sabe qué trozos de código son enteros, cuáles son cadenas de texto, etc, por lo cual es posible que una gran cantidad de errores sean capturados en tiempo de compilación. Por ejemplo, si se desea sumar un número y una cadena de texto, el compilador generará un error al detectar que se quiere aplicar la operación de suma a un valor incorrecto. Por lo anterior se dice que HASKELL es un lenguaje *estáticamente tipado*.

El polimorfismo en HASKELL es paramétrico, lo cual se refiere a que en la definición de una función, el tipo puede declararse utilizando variables de tipo, por ejemplo, la función identidad Id se declara como de tipo $a \to a$, donde a es una variable de tipo. De esta manera en la aplicación $Id\ 2$, la función Id tiene tipo de $Int \to Int$, el cual resulta de la sustitución de la variable a con el tipo concreto Int. El concepto de polimorfismo proporciona una herramienta poderosa de expresividad, al permitir la descripción de una familia, posiblemente infinita como en el caso de Id, de funciones mediante una única declaración.

En cualquier lenguaje de programación las estructuras de dato juegan un papel importante para la solución eficiente a una gran variedad de problemas. El propósito principal de este trabajo consiste en implementar estructuras de datos funcionales, por lo que consideramos conveniente recordar ahora el mecanismo usual de definición de tipos de datos algebraicos en HASKELL.

La forma de definir en HASKELL un tipo de datos algebraico es a través de la declaración `data`, la cual necesita de un nombre para el tipo T que inicie con mayúscula, junto con parámetros de tipo $a_1, \ldots, a_n$, seguido de nombres de constructores $\texttt{cons}_1, \ldots, \texttt{cons}_\texttt{m}$, que a su vez dependen de ciertos parámetros de tipo b_{ik}. La forma general de una declaración `data` es:

$$\begin{array}{rcl} \texttt{data}\ T\ a_1 \ldots a_n & = & \texttt{cons}_1\ b_{11} \ldots b_{1n_1} \\ & | & \texttt{cons}_2\ b_{21} \ldots b_{2n_2} \\ & & . \\ & & . \\ & & . \\ & | & \texttt{cons}_m\ b_{m1} \ldots b_{mn_m} \end{array}$$

De esta forma se declara un constructor de tipo T y funciones constructoras $\texttt{cons}_i$, tales que:

$$\texttt{cons}_i :: b_{i1} \to \ldots b_{in_i} \to T\ a_1 \ldots a_n$$

Estas funciones son llamadas así permiten la construcción de elementos de T.

Dos casos particulares de la definición `data` son:

- Cuando solo hay un constructor con uno o varios parámetros, es decir, $m = 1$. Estos tipos son llamados *producto o tuplas*, y su forma general es:

  $$\texttt{data}\ T\ a_1, \ldots, a_n = \texttt{cons}_1\ b_{11} \ldots\ b_{1n_1}$$

  Por ejemplo:

  ```
  data NumerosComplejos = Complejo Float Float
  ```

- Cuando no hay parámetros en la declaración `data`, es decir, $n = n_i = 0$. Los tipos se llaman *enumerados*, porque justamente es posible enumerar sus elementos y por lo tanto son tipos finitos:

  $$\texttt{data}\ T = \texttt{cons}_1 \mid \texttt{cons}_2 \mid \ \ldots\ \mid \texttt{cons}_m$$

  Por ejemplo:

  ```
  data Movimiento = Izquierda | Derecha | Arriba | Abajo
  ```

  ```
  data Bool = False | True
  ```

También es posible construir tipos de datos en donde los parámetros de un constructor incluyen el tipo que se está definiendo, estos son llamados tipos recursivos. El ejemplo más común es la implementación de listas finitas, en donde es posible almacenar un número arbitrario de elementos. La definición en HASKELL de esta estructura de datos queda como sigue:

```
data List a = Nil | Cons a (List a)
```

La estructura `List` depende de un parámetro de tipo `a`, que corresponde al tipo a almacenar. El primer constructor se llama `Nil`, no tiene parámetros de tipo y representa a una constante que denota a la lista vacía, mientras que el segundo constructor `Cons` recibe dos parámetros de entrada de tipo `a` y `List a` respectivamente y corresponde a la operación de agregar un elemento al inicio de una lista dada.

En adelante utilizaremos conceptos adicionales de HASKELL que consideramos conocidos, si se desea profundizar al respecto recomendamos consultar el libro en [5]

1.2. Estructuras de datos en lenguajes funcionales

Puesto que el objetivo fundamental de este trabajo es desarrollar estructuras de datos funcionales, consideramos adecuado mencionar brevemente algunas características de estas estructuras que no son compartidas por su contraparte en lenguajes no funcionales.

De acuerdo a [6], existen diferencias importantes entre los lenguajes de programación funcional y los lenguajes de programación imperativos u orientada a objetos, tales como C o Java. Estas disimilitudes pueden influir en la implementación de las estructuras de datos, las más destacadas son inmutabilidad, recursión, recolección de basura y coincidencia de patrones (pattern matching).

- *Inmutabilidad:* En los lenguajes funcionales puros, las variables y registros no pueden ser modificados o actualizados, únicamente pueden ser creados. Las estructuras de datos en lenguajes imperativos dependen fuertemente de la asignación o mutación de variables, por lo que resulta inadecuado adaptar esta clase de implementaciones al ámbito funcional.
- *Recursión:* Los lenguajes funcionales no incluyen estructuras de control tales como los ciclos *while* o *for*, puesto que esta clase de instrucciones requiere mutar la variable o contador del ciclo. En su lugar los programadores funcionales deben usar distintos principios de recursión en sus implementaciones.

- *Recolección de basura*: Dado que las entidades en un lenguaje funcional son inmutables, se comparten más ampliamente que en un lenguaje estructural, es decir, la definición de una estructura particular puede ser utilizada para posteriores definiciones o procedimientos. Debido a ésto un lenguaje funcional depende ampliamente de la recolección automática de basura, pues ésta es la única forma segura de liberar espacio de memoria.

- *Coincidencia de patrones (pattern matching):* La coincidencia de patrones es un método de definición de funciones por casos, de tal forma que cada uno de ellos corresponde a una de las posibles maneras de construir la estructura. Este método no es nativo de todos los lenguajes funcionales, pero si está disponible permite la definición concisa y elegante de estructuras de datos y funciones relacionadas.

Con esto terminamos la revisión de los conceptos básicos de programación funcional en Haskell necesarios para el resto del trabajo. En los siguientes capítulos discutimos los conceptos matemáticos de multiconjunto y funtor, que serán indispensables para lograr nuestros objetivos.

Capítulo 2

Multiconjuntos

Nuestro propósito es definir estructuras de datos que cumplan ciertas restricciones, para lo cual nos serviremos del concepto de multiconjunto, que discutimos en este capítulo.

2.1. Definición y propiedades

Intuitivamente un multiconjunto o bolsa A es una colección de objetos desordenados, que a diferencia de los conjuntos, puede contener copias o duplicados. Es decir, pueden existir multiples ocurrencias de un objeto dentro de un multiconjunto.

Los multiconjuntos se pueden representar de diversas maneras, como son:

- *Forma Multiplicativa :* Esta forma es similar a la definición usual de conjuntos por extensión, pero utilizamos corchetes o llaves dobles para hacer la distinción, por ejemplo, un multiconjunto A que tiene una presencia de a, dos presencias de b, y tres presencias de c, puede escribirse como $A = \{\!\{a, b, b, c, c, c\}\!\}$ o bien $A = [a, b, b, c, c, c]$.

- *Multiconjuntos como funciones de valor numérico :* En esta representación un multiconjunto A es una función cuyo dominio incluye a los elementos de A y cuyo contradominio es algún conjunto de números, usualmente el conjunto de los números naturales $\mathbb{N}$. Por ejemplo, el multiconjunto A $= [x, y, y, z, z, z]$, se presenta mediante la función α: $\{x, y, z\} \to \mathbb{N}$ definida por:

$$\alpha(t) = \begin{cases} 1, & \text{si } t = x \\ 2, & \text{si } t = y \\ 3, & \text{si } t = z \\ 0, & \text{en otro caso} \end{cases}$$

La definición de multiconjuntos adecuada para nuestros propósitos es la siguiente:

Definición 2.1 *Sea T un conjunto fijo no vacío. Un multiconjunto A sobre T es un par $A = \langle \mathbb{A}, m_A \rangle$, donde $\mathbb{A} \subseteq T$ es un conjunto y m_A: $T \to \mathbb{N}_\infty$, donde $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$, es una función llamada la función de multiplicidad de A, la cual cumple que para toda x, $x \in \mathbb{A}$ si y solo si $m_A(x) \neq 0$.*

Es importante observar que esta definición incluye la posibilidad de construir no solo multiconjuntos infinitos, sino también multiconjuntos en donde un elemento puede ocurrir un número infinito de veces. Veamos un par de ejemplos:

- A = $\{\!\{1,2,2,3,3,3, \ldots\}\!\}$, se define como A = $\langle \mathbb{N}^+, m_A \rangle$, donde:

$$m_A(n) \quad = \quad n$$

- A = $\{\!\{1,1,1, \ldots\}\!\}$, se define como A = $\langle 1, m_A \rangle$, donde:

$$m_A(1) \quad = \quad \infty$$

A continuación definimos algunos conceptos relevantes relacionados con multiconjuntos sobre un conjunto fijo T.

Definición 2.2 *Sea $A = \langle \mathbb{A}, m_A \rangle$. La relación de pertenencia a A se define como $x \in A$ si y solo si $x \in \mathbb{A}$*

De igual forma que los conjuntos, los multiconjuntos pueden ser enumerados, listando su contenido. Por ejemplo: $M = \{\!\{1,1,2,3\}\!\}$ es un multiconjunto sobre los números naturales $\mathbb{N}$, M contiene el valor 1 dos veces, el valor 2 una vez y el valor 3 una vez, por lo cual $\{\!\{1,1,2,3\}\!\} \neq \{\!\{1,2,3\}\!\}$, sin embargo, se tiene que: $\{\!\{1,1,2,3\}\!\} = \{\!\{1,2,1,3\}\!\}$.

Otra manera de definir multiconjuntos es por comprensión de acuerdo a la siguiente definición:

Definición 2.3 *Sea $A = \langle \mathbb{A}, m_A \rangle$ un multiconjunto. La notación:*

$$S = \{\!\{ x \in A \mid P(x) \}\!\}$$

define al multiconjunto $S = \langle \mathbb{S}, m_S \rangle$, donde $\mathbb{S} = \{ x \in \{\!\{A\}\!\} \mid P(x) \}$ y $m_S(y) = m_A(y)$, si $y \in \mathbb{S}$

La igualdad entre multiconjuntos se define de la siguiente forma: $A = B$, *si y solo si* $m_A(x) = m_B(x)$, *para todo* $x \in T$, conjunto base de A y B. De ésto se desprende la siguiente definición de submulticonjuntos:

Definición 2.4 *Sean $A = \langle \mathbb{A}, m_A \rangle$ y $B = \langle \mathbb{B}, m_B \rangle$ dos multiconjuntos sobre T . Decimos que A es submulticonjunto de B, denotado como $A \subseteq B$ ó $B \supseteq A$, si y solo si $m_A(x) \leq m_B(x)$ para toda $x \in T$. Además, si $A \subseteq B$ y $A \neq B$, entonces A es llamado submulticonjunto propio de B.*

Es posible definir la unión entre multiconjuntos a través de su multiplicidad:

Definición 2.5 *Sean $A = \langle \mathbb{A}, m_A \rangle$, $B = \langle \mathbb{B}, m_B \rangle$ multiconjuntos sobre T. Definimos la unión de A y B, como $A \uplus B = \langle \mathbb{A} \uplus \mathbb{B}, m_{A \uplus B}(x) \rangle$, donde para toda $x \in \mathbb{A} \uplus \mathbb{B}$, $m_{A \uplus B}(x) = m_A(x) + m_B(x)$.*

A continuación mostramos algunas propiedades de la unión de multiconjuntos:

Lema 2.1 *Sean $A = \langle \mathbb{A}, m_A \rangle$, $B = \langle \mathbb{B}, m_B \rangle$ y $C = \langle \mathbb{C}, m_C \rangle$ multiconjuntos sobre T. Se cumplen las siguientes propiedades de la operación unión:*

1. $A \uplus B = B \uplus A$
2. $A \uplus (B \uplus C) = (A \uplus B) \uplus C$
3. $\emptyset \uplus A = A$

Demostración.

1. *P.D.* $\forall\, x \in T\ (m_{A \uplus B}(x) = m_{B \uplus A}(x))$
   Sea $x \in T$,

$$\begin{aligned} m_{A \uplus B}(x) &= m_A(x) + m_B(x) \\ &= m_B(x) + m_A(x) \\ &= m_{B \uplus A}(x) \end{aligned}$$

⊣

2. *P.D.* $\forall\ x \in\ T\ (m_{A \uplus (B \uplus C)}\ (x) = m_{(A \uplus B) \uplus C}\ (x))$
   Sea $x \in T$,

$$\begin{aligned} m_{A \uplus (B \uplus C)}(x) &= m_A(x) + (m_{B \uplus C}(x)) \\ &= m_A(x) + (m_B(x) + m_C(x)) \\ &= (m_A(x) + m_B(x)) + m_C(x) \\ &= m_{(A \uplus B)+C}(x) \\ &= m_{(A \uplus B) \uplus C}(x) \end{aligned}$$

$\dashv$

Existen diferentes áreas en las cuales es relevante la estructura de multiconjuntos, principalmente en Matemáticas y Ciencias de la Computación.

En Matématicas existe una aportación descubierta por Knuth: La factorización en primos de un entero positivo n puede representarse como un multiconjunto F sobre el conjunto de números primos. Como cada entero positivo puede ser factorizado de manera única por números primos, se obtiene una biyección entre los enteros positivos y el conjunto de multiconjuntos finitos de números primos. Por ejemplo, si n = $2^2 \cdot 3^3 \cdot 17$, le corresponde el multiconjunto F = [2,2,3,3,3,17]. Otra aportación importante dentro del área de las Matématicas, es la correspondencia entre un polinomio mónico sobre los números complejos y el multiconjunto único que contiene sus raíces.

Dentro de las Ciencias de la Computación se ha estudiado a los multiconjuntos como un tipo de dato abstracto, ésto lo hace una herramienta de modelado adecuado para resolver o aplicar en diversos tipos de problemas de la vida real. Por ejemplo, Ross y Stoyanovich estudiaron multiconjuntos de cardinalidad acotada en sistemas de bases de datos para superar problemas de consistencia y rendimiento, sufridos por las representaciones convencionales en bases de datos relacionales.

Para conocer otras maneras de definir multiconjuntos, así como sus aplicaciones, recomendamos revisar el artículo en [7].

La manera en que nosotros usaremos multiconjuntos se desarrollará en las siguientes secciones, en particular nos interesan algunas propiedades y operaciones de multiconjuntos de números naturales discutidas a continuación.

2.2. Multiconjuntos de números naturales

En adelante trataremos únicamente con multiconjuntos de números naturales, los cuales además de las operaciones generales ya definidas, cuentan con operaciones aritméticas heredadas de aquellas para los números naturales. En particular utilizaremos la suma y producto de multiconjuntos definidas como sigue:

Definición 2.6 *Sean M_1 y M_2 multiconjuntos de números naturales. Definimos las operaciones de suma y multiplicación como sigue:*

$$M_1 + M_2 = \{\!\{ m_i + m_j \mid \forall\, m_i \in M_1 \ y \ \forall\, m_j \in M_2 \}\!\}$$

$$M_1 * M_2 = \{\!\{ m_i * m_j \mid \forall\, m_i \in M_1 \ y \ \forall\, m_j \in M_2 \}\!\}$$

En esta clase de definiciones se deben respetar las multiplicidades de los multiconjuntos originales, por ejemplo: si $M_1 = \{\!\{2, 4, 2, 1\}\!\}$ y $M_2 = \{\!\{3, 3, 1\}\!\}$, entonces $M_1 + M_2 = \{\!\{5, 5, 3, 7, 7, 5, 5, 5, 3, 4, 4, 2\}\!\}$.

Presentamos ahora algunas propiedades de las operaciones ariméticas de multiconjuntos, que nos serán de utilidad más adelante:

Lema 2.2 *Sean $A = \langle \mathbb{A}, m_A \rangle$, $B = \langle \mathbb{B}, m_B \rangle$ y $C = \langle \mathbb{C}, m_C \rangle$ multiconjuntos sobre los números naturales. Se cumplen las siguientes propiedades de suma entre multiconjuntos:*

1. $A + B = B + A$
2. $A + (B + C) = (A + B) + C$
3. $\{\!\{0\}\!\} + A = A$
4. $\emptyset + A = \emptyset$
5. $(A \uplus B) + C = (A + C) \uplus (B + C)$

Demostración.

1.
$$\begin{aligned} A + B &= \{\!\{a + b \mid a \in A, b \in B\}\!\} \\ &= \{\!\{b + a \mid a \in A, b \in B\}\!\} \\ &= B + A \end{aligned}$$

⊣

2.

$$\begin{aligned} A + (B + C) &= \{\!\{a + k \mid a \in A, k \in (B + C)\}\!\} \\ &= \{\!\{a + k \mid a \in A, k \in \{\!\{b + c \mid b \in B, c \in C\}\!\}\}\!\} \\ &= \{\!\{a + (b + c) \mid a \in A, b \in B, c \in C\}\!\} \\ &= \{\!\{(a + b) + c \mid a \in A, b \in B, c \in C\}\!\} \\ &= \{\!\{k + c \mid k \in \{\!\{a + b \mid a \in A, b \in B\}\!\}, c \in C\}\!\} \\ &= \{\!\{k + c \mid k \in A + B, c \in C\}\!\} \\ &= (A + B) + C \end{aligned}$$

⊣

3.

$$\begin{aligned} \{\!\{0\}\!\} + A &= \{\!\{0 + a \mid a \in A\}\!\} \\ &= \{\!\{a \mid a \in A\}\!\} \\ &= A \end{aligned}$$

⊣

Veamos ahora algunas propiedades generales del producto:

Lema 2.3 *Sean $A = \langle \mathbb{A}, m_A \rangle$, $B = \langle \mathbb{B}, m_B \rangle$ y $C = \langle \mathbb{C}, m_C \rangle$ multiconjuntos sobre los números naturales. Se cumplen las siguientes propiedades:*

1. $\{\!\{1\}\!\} * A = A$
2. $A * \{\!\{1\}\!\} = A$
3. $A * (B * C) = (A * B) * C$
4. $(A \uplus B) * C = A * C \uplus B * C$
5. $\emptyset * A = \emptyset$

Demostración.

1.

$$\begin{aligned} \{\!\{1\}\!\} * A &= \{\!\{x * y \mid x \in \{\!\{1\}\!\}, y \in A\}\!\} \\ &= \{\!\{1 * y \mid y \in A\}\!\} \\ &= \{\!\{y \mid y \in A\}\!\} \\ &= A \end{aligned}$$

⊣

2.
$$\begin{aligned} A * \{\!\{1\}\!\} &= \{\!\{y * 1 \mid y \in A\}\!\} \\ &= \{\!\{y \mid y \in A\}\!\} \\ &= A \end{aligned}$$

⊣

3.
$$\begin{aligned} A * (B * C) &= \{\!\{a * x \mid a \in A, x \in B * C\}\!\} \\ &= \{\!\{a * x \mid a \in A, x \in \{\!\{b * c \mid b \in B, c \in C\}\!\}\}\!\} \\ &= \{\!\{a * (b * c) \mid a \in A, b \in B, c \in C\}\!\} \\ &= \{\!\{(a * b) * c \mid a \in A, b \in B, c \in C\}\!\} \\ &= \{\!\{x * c \mid x \in A * B, c \in C\}\!\} \\ &= (A * B) * C \end{aligned}$$

⊣

Existe otras propiedades importantes del producto que involucran a una clase particular de multiconjuntos.

Definición 2.7 *Un multiconjunto $A = \langle \mathbb{A}, m_A \rangle$ es simple si y solo si $\mathbb{A} = \{a\}$ y $m_A(a) = 1$, es decir un multiconjunto es simple si tiene un único elemento el cual figura un sola vez.*
Si A es simple lo denotamos como $A = \{\!\{a\}\!\}$. Más aún, en adelante convenimos en denotar con letras minúsculas $a, b, c \ldots$, a los multiconjuntos simples $a = \{\!\{a\}\!\}, b = \{\!\{b\}\!\}, c = \{\!\{c\}\!\}$.

Definición 2.8 *Sean A y B multiconjuntos. Se dice que el producto $A * B$ es admisible si y solo si B es un multiconjunto simple.*

Por ejemplo el producto $\mathbb{N} * \{\!\{2\}\!\}$ es admisible, mientras que el producto $\{\!\{2\}\!\} * \mathbb{N}$ no lo es.

Los multiconjuntos simples tienen las siguientes propiedades respecto al producto.

Lema 2.4 *Sean a, b y c multiconjuntos simples. Se cumplen las siguientes propiedades:*

1. $(A + B) * c = A * c + B * c$
2. $a * b = b * a$

3. $\{\!\{0\}\!\} * a = \{\!\{0\}\!\}$

Demostración.

1.
$$\begin{aligned}(A+B)*c &= \{\!\{x*c \mid x \in A+B\}\!\} \\ &= \{\!\{(a+b)*c \mid a \in A, b \in B\}\!\} \\ &= \{\!\{(a*c)+(b*c) \mid a \in A, b \in B\}\!\} \\ &= \{\!\{x+y \mid x \in A*c, y \in B*c\}\!\} \\ &= (A*c)+(B*c)\end{aligned}$$

⊣

Un ejemplo de que la propiedad 1 y 3 propiedades no se cumplen cuando los multiconjuntos no son simples, es el siguiente:

Sea $A = \{\!\{2,1\}\!\}$, $B = \{\!\{1,4\}\!\}$ y $C = \{\!\{1,2\}\!\}$

- $$\begin{aligned}&(A+B)*C \\ &\quad= (\{\!\{2,1\}\!\}+\{\!\{2,4\}\!\})*\{\!\{1,2\}\!\} \\ &\quad= \{\!\{4,6,3,5\}\!\}*\{\!\{1,2\}\!\} \\ &\quad= \{\!\{4,8,6,12,3,6,5,10\}\!\}\end{aligned}$$

  $$\begin{aligned}&(A*C)+(B*C) \\ &\quad= (\{\!\{2,1\}\!\}*\{\!\{1,2\}\!\})+(\{\!\{1,4\}\!\}*\{\!\{1,2\}\!\}) \\ &\quad= \{\!\{2,4,1,2\}\!\}+\{\!\{1,2,4,8\}\!\} \\ &\quad= \{\!\{3,4,6,10,4,6,8,12,2,3,5,9,3,4,6,10\}\!\}\end{aligned}$$

  Por lo tanto $(A+B)*C \neq (A*C)+(B*C)$

- $$\begin{aligned}&\{\!\{0\}\!\}*A \\ &\quad= \{\!\{0\}\!\}*\{\!\{2,1\}\!\} \\ &\quad= \{\!\{0,0\}\!\}*\{\!\{1,2\}\!\}\end{aligned}$$

  Por lo tanto $\{\!\{0\}\!\}*A \neq \{\!\{0\}\!\}$

En el caso de la propiedad 2 se restringe a los multiconjuntos simples debido a la correspondencia entre multiconjuntos y funtores que veremos en la sección 3.3

Nuestro propósito es utilizar multiconjuntos y ecuaciones de multiconjuntos para definir formalmente las restricciones de tamaño o estructura dadas en la especificación de un tipo de dato, pero antes definimos un mecanismo sintáctico para denotar multiconjuntos, el cual simplificará la presentación. Consideraremos únicamente multiconjuntos definidos mediante la siguiente gramática:

$$\mathrm{M} := \emptyset \mid 0 \mid 1 \mid \mathrm{M} + \mathrm{M} \mid \mathrm{M} * \mathrm{M} \mid \mathrm{M} \uplus \mathrm{M}$$

Esta gramática corresponde a expresiones cuyo significado $[\![\cdot]\!]$ es el esperado:

$$\begin{array}{rcl} [\![\emptyset]\!] & = & \emptyset \\ [\![0]\!] & = & \{\!\{0\}\!\} \\ [\![1]\!] & = & \{\!\{1\}\!\} \\ [\![M_1 \uplus M_2]\!] & = & [\![M_1]\!] \uplus [\![M_2]\!] \\ [\![M_1 * M_2]\!] & = & [\![M_1]\!] * [\![M_2]\!] \\ [\![M_1 + M_2]\!] & = & [\![M_1]\!] + [\![M_2]\!] \end{array}$$

En adelante también utilizaremos la notación n para denotar al multiconjunto $\{\!\{\mathrm{n}\}\!\}$. Esta es en realidad una abreviatura para la expresión $n = 1 +1 + \ldots + 1$, con n presencias de la expresión 1.

Por ejemplo la expresión $M = (1 + 3) \uplus (4 * 2)$ denota al multiconjunto $[\![M]\!] = \{\!\{4, 8\}\!\}$, puesto que

$$\begin{array}{rcl} [\![M]\!] & = & [\![(1 + 3) \uplus (4 * 2)]\!] \\ & = & [\![1 + 3]\!] \uplus [\![4 * 2]\!] \\ & = & [\![1]\!] + [\![3]\!] \uplus [\![4]\!] * [\![2]\!] \\ & = & \{\!\{1\}\!\} + \{\!\{3\}\!\} \uplus \{\!\{4\}\!\} * \{\!\{2\}\!\} \\ & = & \{\!\{4\}\!\} \uplus \{\!\{8\}\!\} \\ & = & \{\!\{4, 8\}\!\} \end{array}$$

2.3. Ecuaciones de multiconjuntos

Nuestro mecanismo de definición de estructuras de datos se basa en el planteamiento y solución de ecuaciones de multiconjuntos, en particular ecuaciones recursivas o de punto fijo. Estas ecuaciones son de la forma $M = N$, donde M y N son expresiones de la gramática anterior.
Como ejemplo considerese las siguientes ecuaciones:

$$\begin{aligned} M_1 &= 1 \uplus M_1 \\ M_2 &= 1 + M_2 \\ M_3 &= 1 * M_3 \end{aligned}$$

El significado y la solución a estas ecuaciones se obtiene de la manera usual, mediante la semántica del mínimo punto fijo, véase [2]. Por ejemplo, la segunda y tercera ecuación tienen como solución al multiconjunto vacío, es decir $M_2 = M_3 = \emptyset$. Por otra parte la solución a la primera ecuación es $M_1 = \{\!\{1, 1, 1, ...\}\!\}$, es decir es el multiconjunto que tiene una infinidad de unos.

Dada una restricción particular en la especificación de un tipo de dato, nuestro objetivo es capturarla mediante una ecuación o sistema de ecuaciones de multiconjuntos. Por ejemplo, si la especificación restringe a que los contenedores en una estructura de datos sean de tamaño arbitrario (sin restricción) impar, par o una potencia de dos, será necesario definir los multiconjuntos de números naturales, impares, pares o potencias de dos mediante una ecuación como las siguientes:

1. Números naturales:

$$N = 0 \uplus (1 + N)$$

2. Números impares:

$$O = 1 \uplus (O + 2)$$

3. Números pares:

$$E = 0 \uplus (E + 2)$$

4. Potencias de dos:

$$P = 1 \uplus (P * 2)$$

Esta clase de ecuaciones de multiconjuntos son parte importante del proceso de manufactura de tipos de datos y corresponden a una definición recursiva de los respectivos multiconjuntos. Por ejemplo, la ecuación que define al multiconjunto de números pares corresponde a la generación recursiva de números pares a partir del cero mediante la operación $(x \mapsto x + 2)$.

La definición de un multiconjunto particular mediante una ecuación, no es única ya que podemos utilizar definiciones directas o bien utilizando otra clase de recursión, por ejemplo:

1. Números impares:

$$O = 1 + N * 2$$

2. Números pares e impares (mutuamente):

$$\begin{aligned} O &= 1 + E \\ E &= 0 \uplus (2 + E) \end{aligned}$$

3. Números naturales perezosos:

$$\begin{aligned} N &= F\ 0 \\ F\ n &= n \uplus F(1 + n) \end{aligned}$$

4. Números cuadrados perezosos:

$$\begin{aligned} S &= F\ 0 \\ F\ n &= n * n \uplus F(1 + n) \end{aligned}$$

Los dos últimos ejemplos utilizan una función auxiliar F para su definición, esta función genera los números de la forma deseada a partir de un número inicial dado, para los naturales Fn genera todos los números naturales a partir de n, mientras que para los cuadrados Fn devuelve todos los números cuadrados a partir de n^2. El adjetivo perezosos se refiere a que estas definiciones solo tienen sentido mediante la estrategia de evaluación perezosa, puesto que la definición de la función F no es recursiva estructural.

En este capítulo hemos desarrollado el concepto de multiconjunto y se han exhibido algunas de sus propiedades. Esta herramienta nos permitirá definir multiconjuntos de números naturales a partir de la restricción de tamaño en la especificación de una estructura de datos. Nuestro siguiente paso es definir formalmente el concepto de estructura de datos o tipo contenedor, mediante el concepto matemático de funtor.

Capítulo 3

Funtores

Existen diversas interpretaciones sobre lo que significa un funtor, pero para nuestro propósito trataremos a los funtores como operadores entre tipos, cuya finalidad es transformar un tipo dado A en un tipo contenedor de elementos de A. El concepto de funtor en matemáticas surge de la teoría de las categorías, rama del álgebra con diversas aplicaciones en la computación teórica que, sin embargo, no utilizaremos aquí.

Un tipo contenedor se caracteriza por almacenar elementos de cierto tipo. No debe importar el tipo de elementos que es posible almacenar en él, ni se debe imponer alguna restricción sobre ellos, más bien debemos definir funciones generales que manipulen a elementos del contenedor, como son: el agregar, actualizar o eliminar algún elemento. De mayor interés resultan las operaciones que transforman los elementos de un contenedor en elementos de otro tipo dentro del contenedor mismo. El ejemplo más familiar e inmediato de funtor es el constructor de listas: podemos almacenar elementos de algún tipo y operar con ellos sin importar realmente de que tipo son, por ejemplo, podemos recorrer los elementos de una lista de árboles, así como de un lista de números naturales para obtener el último elemento o aplicar una transformación de tipos a cada uno de los elementos.

Formalmente, un funtor conserva la estructura entre dos *categorías*. No ahondaremos en la definición de este concepto matemático, para nuestro propósito basta decir que una *categoría* es una colección de tipos y funciones entre ellos, por lo que un funtor es un operador, que además de transformar tipos, debe transformar de manera adecuada las funciones entre los mismos.

En HASKELL, se definen a los operadores que son funtores mediante la clase `Functor`, cuya definición es:

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

En esta definición `f` es un operador que transforma tipos en tipos, mientras `fmap` se encarga de transformar una función arbitraria *h:* $a \to b$ en una función *(fmap h) ::* $f\ a \to f\ b$, la cual transforma contenedores de a en contenedores de b.

En el caso de las listas, el funtor `f` se denota con [ ] y la función `fmap` no es más que la función `map` para listas, con lo cual se crea una instancia de la clase `Functor` como sigue:

```
instance Functor [ ] where
    fmap =  map
```

Obsérvese que `map` es una función que está definida en el preludio de HASKELL y cuyo tipo es:

```
map :: (a -> b) -> [a] -> [b]
```

Un ejemplo de la aplicación de la función `fmap` en el caso del funtor lista, es el siguiente:

```
* Main> (fmap (2 *)) [1, 4, 5, 6, 3]
[2, 8, 10, 12, 6]
```

En este caso específico, la transformación consiste en duplicar los elementos de la lista, la signatura de `fmap` está dada por:

```
 fmap :: (Int -> Int) -> ([Int] -> [Int])
```

y la regla de transformación está definida como sigue:

```
h:: Int -> Int
   h x = 2 * x
```

Otro ejemplo relacionado con listas consiste en transformar sus elementos en pares, en donde cada uno contienen a cada elemento y a su triple. Por ejemplo:

```
Main> (fmap triple) [1, 4, 5]
[(1,3), (4,12), (5,15) ]
```

Por lo cual, la signatura de `fmap` queda como sigue:

```
 fmap :: (Int -> (Int,Int)) -> ([Int] -> [(Int,Int)])
```

y la regla de trasformación es:

```
triple:: Int -> (Int,Int)
   h x = (x, 3*x)
```

Otro ejemplo de tipo contenedor son los árboles binarios, cuya definición en HASKELL es:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a ) deriving(Show)
```

Podemos convertir al operador `Tree` en un tipo contenedor creando una instancia de la clase `Funtor`, como se muestra a continuación:

```
instance Functor Tree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch left right) = Branch (fmap f left)
                                        (fmap f right)
```

Un ejemplo de aplicación de la función `fmap` es el siguiente:

```
Main> (fmap (2 *)) (Branch(Branch(Leaf 1)(Leaf 2))
                          (Branch(Leaf 3)(Leaf 4)))


Branch(Branch(Leaf 2)(Leaf 4)) (Branch(Leaf 6)(Leaf 8))
```

Aquí estamos transformando un árbol duplicando el valor de sus nodos, la signatura particular de `fmap` está dada por:

```
 fmap :: (Int -> Int) -> (Tree Int -> Tree Int)
```

y la regla de transformación está definida de la siguiente forma:

```
h:: Int -> Int
   h x = 2 * x
```

Otra transformación consiste en sustituir el valor de las hojas en un árbol de listas, por su longitud. Por ejemplo:

```
Main> (fmap length) (Branch(Branch(Leaf [5,2])(Leaf [1]))
                    (Branch(Leaf [4,3,2])(Leaf [4,5])))


Branch(Branch(Leaf 2)(Leaf 1)) (Branch(Leaf 3)(Leaf 2))
```

En esté caso, la signatura de `fmap` queda definida como sigue:

```
fmap :: ([Int] -> Int) -> (Tree [Int] -> Tree Int)
```

No todo operador que transforma tipos en tipos puede ser considerado un funtor. Para que un funtor `f` cumpla la idea de un contenedor, es importante definir correctamente la función `fmap`, por lo cual es necesario asegurar que su definición cumpla la siguientes leyes:

- Primera ley funtorial:

  ```
  fmap id = id
  ```

  donde `id` es la función identidad. Esta ley establece que `fmap` transforma la función `id`, de tipo $a \to a$, en la función `id`, de tipo $f\ a \to f\ a$. Debido a la defínción de la función identidad, se tiene que al aplicar `id` sobre un contenedor, se regresará el mismo contenedor sin alteraciones.

- Segunda ley funtorial:

  ```
  fmap (f · g) = fmap f · fmap g
  ```

  Esta ley pide que `fmap` asocie a la composición `f · g` la composición `fmap f · fmap g`. Como consecuencia, por la definición de composición, sucede que aplicar la transformación `f · g` a un contenedor es equivalente a aplicarle primero la transformación `g` a dicho contenedor para posteriormente aplicarle la transformación `f` al resultado.

Como ejemplo, considérese un árbol que almacena listas de naturales, deseando obtener el árbol que contiene la suma de los elementos duplicados de cada lista contenida en los nodos, cuyo resultado será un árbol que almacena números naturales. Esta tarea puede realizarse en dos pasos:

- Transformar el árbol dado duplicando los elementos de cada lista.
- Transformar el resultado del paso anterior sustituyendo cada lista por la suma de sus elementos.

En la Figura 3.1, se muestra un ejemplo de esto. Este procedimiento corresponde a la composición de dos transformaciones mediante `fmap`, a saber

```
fmap h2 (fmap h1 T)
```

donde,

```
h1 x = 2 * x
h2 xs = sum xs
```

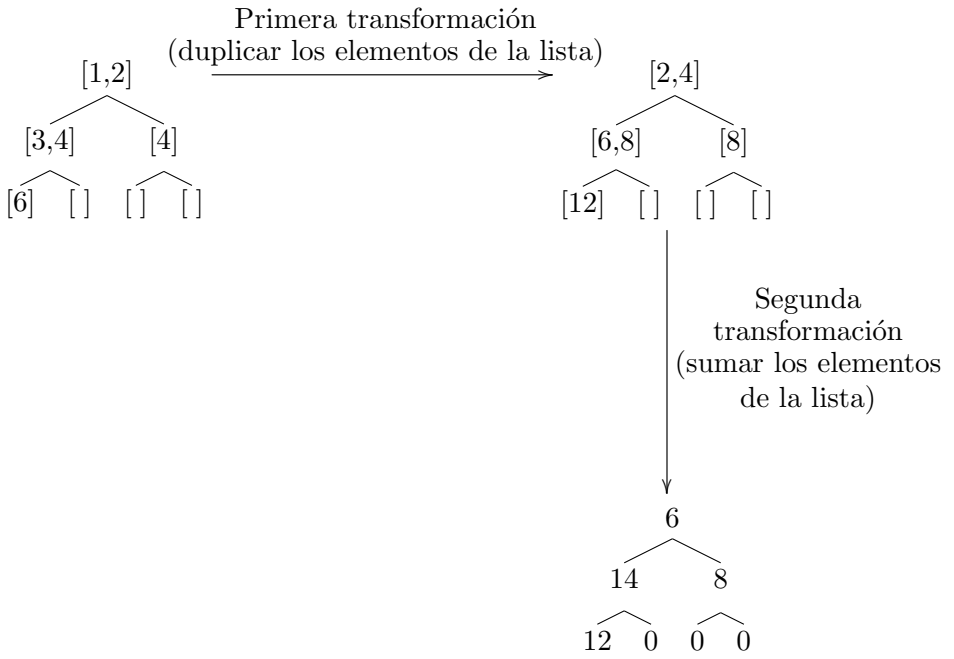

Figura 3.1: Ejemplo de duplicar y sumar los elementos de las listas almacenadas en T

Este procedimiento resulta ineficiente puesto que se está construyendo un árbol intermedio (el árbol resultante de duplicar los elementos de cada lista). Es posible evitar la construcción de este árbol componiendo las dos reglas de transformación ($h1$ y $h2$) de acuerdo a la ley funtorial definida arriba, de la siguiente manera:

```
fmap (h2 . h1) T
```

En este caso, las dos transformaciones se componen para obtener una única transformación consistente en duplicar los elementos de una lista para después sumarlos, con lo cual no se genera el árbol intermedio.

3.1. Funtores en Haskell

Ahora que ya contamos con algunos ejemplos de funtor, nos interesa definir operaciones que construyan funtores a partir de otros funtores ya dados.

Las operaciones que nos interesan se pueden definir mediante la siguiente gramática:

$$F ::= KVoid \mid KUnit \mid Id \mid (F \mid F) \mid (F \times F) \mid (F \cdot F)$$

En adelante nos interesarán únicamente los funtores generados por esta gramática. Analicemos cada caso: *KVoid* es el funtor constante dado por *KVoid a = Void*, este operador transforma cualquier tipo dado *a* en el tipo vacío *Void*; análogamente *KUnit* es el funtor que transforma cualquier tipo *a* en el tipo *Unit*, el cual contiene un solo elemento [1]. Estos dos funtores corresponden a la creación del contenedor vacío de elementos de *a* y de un contenedor abstracto que guarda un elemento fijo que no es posible cambiar. Aparentemente estos funtores no son de utilidad, lo cual se mostrará falso más adelante. El funtor identidad se denota como *Id* y está dado por *Id a = a*. Los siguientes casos definen nuevos funtores a partir de otros, el funtor composición lo denotamos con *F1 · F2*, dado por *(F1 · F2) a = F1 (F2 a)* y corresponde a la creación de contenedores de contenedores, por ejemplo árboles de listas. La suma y el producto están definidos como *(F1 | F2) a = F1 a | F2 a* y *(F1 × F2) a = F1 a × F2 b* y corresponden a la unión de contenedores, por ejemplo listas o árboles; y al encapsulamiento de dos contenedores en un tercero, por ejemplo pares de listas y árboles.

La gramática anterior corresponde a los siguientes mecanismos de definición de tipos en HASKELL:

```
data Void
data Unit = ()
data Either a1 a2 = Left a1 | Rigth a2
data (a1, a2) = (a1, a2)
newtype KVoid a  = Void
newtype KUnit a  = Unit
newtype FId a = CId a
newtype FSum t1 t2 a = CSum (Either (t1 a)(t2 a))
newtype FProd t1 t2 a = CProd (t1 a, t2 a)
newtype FComp t1 t2 a = CComp (t1(t2 a))
```

La definición de estos tipos no garantizan que sean funtores, debemos instanciarlos a la clase funtor. Ahora, construimos la función `fmap` para cada una de las definiciones anteriores:

[1] Consideramos importante mencionar aquí que el tipo Unit es lo que en muchos lenguajes de programación no funcionales se conoce como el tipo Void

- Funtor Vacío

```
fmap :: (a -> b) -> KVoid a -> KVoid b
fmap _  x = x
```

- Funtor Unitario

```
fmap :: (a -> b) -> KUnit a -> KUnit b
fmap _  x = x
```

- Funtor Identidad

```
fmap :: (a -> b) -> FId a ->  FId b
fmap f (CId x) = CId (f x)
```

- Funtor Suma

```
fmap :: (a -> b) -> Fsum t1 t2 a ->  Fsum t1 t2 b
fmap f (CSum (Left x)) = CSum (Left (fmap1 f x))
fmap f (CSum (Right x)) = CSum (Right (fmap2 f x))
```

donde,

```
fmap1 :: (a -> b) -> t1 a ->  t1 b
fmap2 :: (a -> b) -> t2 a ->  t2 b
```

- Funtor Producto

```
fmap :: (a -> b) -> FProd t1 t2  a ->  FProd t1 t2 b
fmap f (CProd (x,y)) = CProd (fmap1 f x, fmap2 f y)
```

donde,

```
fmap1 :: (a -> b) -> t1 a ->  t1 b
fmap2 :: (a -> b) -> t2 a ->  t2 b
```

- Funtor Composición

```
fmap :: (a -> b) -> FComp t1 t2  a ->  FComp t1 t2 b
fmap f (CComp x) = CComp (fmap1 (fmap2 f) x)
```

donde,

```
fmap1 :: (t2 a -> t2 b) -> t1(t2 a) ->  t1(t2 b)
fmap2 :: (a -> b) -> t2 a ->  t2 b
```

A continuación verificamos que las definiciones anteriores en efecto cumplen con las leyes funtoriales.

Proposición 3.1 *KVoid, KUnit y Fid cumplen las leyes funtoriales*

Demostración. Para los funtores *KVoid* y *KUnit* la prueba es la misma y se sigue directamente de la definición de `fmap`.

Para el funtor identidad procedemos como sigue:

- Primera ley funtorial.

  ```
  fmap id (CId x) = CId (id x)
                  = CId x
  ```

  Por lo tanto, `fmap id = id`

- Segunda ley funtorial.

  ```
  fmap (f · g) (CId x)= CId ((f · g) x)
                      = CId  (f (g x))
                      = fmap f (CId (g x))
                      = fmap f (fmap g (CId x))
                      = (fmap f · fmap g) (CId x)
  ```

  Por lo tanto, `fmap (f· g) = (fmap f · fmap g)`

⊣

Proposición 3.2 *Sean f_1 y f_2 funtores entonces FSum f_1 f_2, FProd f_1 f_2 y FComp f_1 f_2 son funtores*

Demostración. Sean f_1 y f_2 funtores y `fmap1`, `fmap2` sus funciones map correspondientes. Demostramos el caso para $FSum$, los otros casos son análogos

- Primera ley funtorial.

```
fmap id (CSum (Left x)) = CSum (Left (fmap1 id x))
                        = CSum (Left x)
```

  El caso que involucra el constructor `Right` se resuelve de manera análoga utilizando `fmap2`.

- Segunda ley funtorial.

```
fmap  (f · g) (CSUm (Right x))
= CSum (Right ( fmap2 (f · g ) x ))
= CSum (Right ( fmap2 f (fmap2 g x)))
= fmap f ( CSum (Right (fmap2 g x)))
= fmap f( fmap g ( CSum ( Right x )))
= (fmap f ·  fmap g) ( CSum · (Right x))
```

  El caso para el constructor `Left` se resuelve de manera análoga.

⊣

Nuestro siguiente objetivo es relacionar los conceptos de multiconjunto y funtor, de tal forma que a partir de un multiconjunto M sea posible definir un constructor de tipos, es decir, un funtor F en donde el tipo Fa tenga exactamente un contenedor de tamaño n, para cada $n \in M$. Para esto necesitamos conceptos de la programación genérica.

3.2. Programación genérica

Una función génerica es aquella que puede definirse mediante recursión en la estructura de los tipos. El ejemplo clásico de una función generica es $size :: Fa \to Int$ la cual cuenta el número de valores de tipo a almacenados en un contenedor de tipo Fa. Esta función se define prácticamente de la misma forma para cada tipo Fa. Pensemos por ejemplo en las definiciones particulares de la función *size* para listas y árboles, en el primer caso, el tamaño de una lista se obtiene sumando 1 al tamaño de su cola, mientras que en el segundo, el tamaño de un árbol se obtiene sumando 1 a la suma de los tamaños de los subárboles. Para poder definir una única función $size :: Fa \to Int$ es necesario hacer recursión sobre los constructores de tipo, que en nuestro caso son los funtores denotados por F. La metodología que permite esta clase de definiciones se conoce como programación genérica o

politípica. Específicamente, las funciones politípicas capturan a familias de funciones polimórficas en una única definición. La programación genérica se utiliza cuando varios algoritmos tienen que ser implementados una y otra vez, únicamente porque los tipos de datos cambian durante el desarrollo o implementación de los programas, pero no la definición del algoritmo. Ejemplos prominentes de esta técnica son las funciones obtenidas mediante las instancias automáticas en HASKELL a ciertas clases, por medio de la instrucción `deriving`. Para una exposición más profunda de la programación genérica sugerimos consultar el artículo en [1]

A continuación definimos algunas funciones genéricas que nos serán de utilidad para lograr la correspondencia entre multiconjuntos y funtores. Empezamos con la definición de la función `sum`, que suma todos los elementos de un contenedor de números naturales. La signatura es la siguiente:

```
sum⟨F⟩:: F Nat -> Nat
```

Obsérvese que el argumento ⟨F⟩ va a variar de acuerdo a la definición del constructor de tipos `F`. La definición genérica es:

```
sum⟨KVoid⟩ x = error
sum⟨KUnit⟩ x = 0
sum⟨Id⟩ x = x
sum⟨F1 | F2⟩ x = case x of { Left x1 -> sum<F1> x1;
                            Right x2 -> sum<F2> x2 }
sum⟨F1  ×  F2⟩ x = sum⟨F1⟩ (fst x) + sum⟨F2⟩(snd x)
sum⟨F1 · F2⟩ x = sum⟨F1⟩(fmap⟨F1⟩ sum⟨F2⟩ x)
```

En el último caso se requiere de la función genérica `fmap`, que definimos a continuación:

```
fmap⟨F⟩ :: (a -> b) -> Fa -> Fb
fmap⟨KUnit⟩ f x = x
fmap⟨KVoid⟩ f x = x
fmap⟨Id⟩ f x = x

fmap⟨F1 | F2⟩ f x = case x of
                      Left x1 -> Left(fmap⟨F1⟩ f x1);
                    | Right x2 -> Right(fmap⟨F2⟩ xf x2)
fmap⟨F1 × F2⟩ x = (mapF⟨F1⟩ f (fst x), mapF⟨F2⟩ f (snd x))
fmap⟨F1· F2⟩ x = fmap⟨F1⟩(fmap⟨F2⟩ f) x
```

Finalmente necesitamos una función genérica `fan` que genera todos los multiconjuntos de tipo `F a` y que guardan un único elemento de tipo `a`. Por ejemplo, $\texttt{fan}\langle List\rangle\ 1$ genera el multiconjunto de todas las listas finitas que contienen como único elemento al número natural 1, es decir, $\texttt{fan}\langle List\rangle\ 1 = \{\!\{Nil, Cons\ 1\ Nil, Cons\ 1(Cons\ 1\ Nil), Cons\ 1(Cons\ 1(Cons\ 1\ Nil)), \ldots\}\!\}$.

La definición de fan es:

```
fan⟨F⟩ :: a -> {|F a|}
fan⟨K Void⟩ x =  ∅
fan⟨K Unit⟩ x = {|()|}
fan⟨Id⟩ x ={|x|}
fan⟨F1 | F2⟩ x = {|Left x1 | x1 <- fan⟨F1⟩ x|} ⊎
                 {|Right x2 | x2 <- fan⟨F2⟩ x|}
fan⟨F1 × F2⟩ x = {|(x1,x2) | x1 <- fan⟨F1⟩ x ; x2 <- fan⟨F2⟩ x|}
```

En el caso de la composición de funtores, si bien podría darse una definición general, por razones que se verán más adelante nos interesa solo el siguiente caso particular:

$\texttt{fan}\langle \texttt{F1} \cdot \texttt{Id}^n\rangle$ `x = {|fan` $\langle$`F1`$\rangle$ `y | y <-` $\texttt{fan}\langle \texttt{Id}^n\rangle$ `x|}`

aquí Id^n denota el funtor $\underbrace{Id \times Id \times \cdots \times Id}_{n-veces}$.

3.3. Correspondencia Funtores-Multiconjuntos

Al incio de este capítulo formalizamos un concepto de tipo contenedor mediante la noción de funtor en Haskell y anteriormente en la sección 2.3 vimos como una restricción particular de los posibles tamaños de los elementos de un tipo contenedor, puede definirse mediante una ecuación de multiconjuntos. El objetivo ahora es relacionar una ecuación de multiconjuntos con un funtor F, que construya tipos contenedores con la restricción deseada, de tal forma que si F corresponde al multiconjunto M, entonces para cualquier tipo a, el tipo Fa contiene por cada elemento $n \in M$, un contenedor de elementos de a de tamaño m. Esto se hará mediante la transformación de multiconjuntos en una ecuación de funtores de acuerdo a la siguiente correspondencia.

Definición 3.1 *Dado un multiconjunto M podemos definir un funtor F a partir de M de acuerdo a la siguiente correspondencia:*

1. *Si* $M = \emptyset$ *entonces* $F = KVoid$
2. *Si* $M = 0$ *entonces* $F = KUnit$
3. *Si* $M = 1$ *entonces* $F = Id$
4. *Si* M_1 *corresponde a* F_1 *y* M_2 *corresponde a* F_2 *entonces:*
   - $M_1 + M_2$ *corresponde a* $F_1 \times F_2$
   - $M_1 * M_2$ *corresponde a* $F_1 \cdot F_2$
   - $M_1 \uplus M_2$ *corresponde a* $F_1 \mid F_2$

Una forma simplificada de ver lo anterior es a través de la siguiente tabla:

M1	M2	$\emptyset$	0	1	M1 + M2	M1 $*$ M2	M1 $\uplus$ M2
F1	F2	K Void	K Unit	Id	F1 $\times$ F2	F1$\cdot$ F2	F1 $\mid$ F2

Los multiconjuntos simples serán de gran importancia en adelante y su funtor correspondiente se obtiene a partir de los casos para los multiconjuntos 0 y 1 utilizando las demás operaciones. Por ejemplo, puesto que $2 = 1 + 1$, el funtor correspondiente al multiconjunto simple 2 resulta ser el funtor Id $\times$ Id.

A continuación mostramos algunos ejemplos en donde M es la ecuación de multiconjuntos y F la ecuación funtorial correspondiente, siguiendo la especificación de la tabla anterior.

1. `M = {{0,0}}`
   `= 0 ⊎ 0`
   `F = KUnit | KUnit`

2. `M = {{1,1}}`
   `= 1 ⊎ 1`
   `F = Id | Id`

3. `M = 1 ⊎ (1 + M)`
   `F = Id | Id × F`

4. `M = 1 ⊎ (M + M)`
   `F = Id | F × F`

Para asegurar la correspondencia de un multiconjunto M con un funtor F, es necesario construir una forma para obtener todos los tamaños posibles de los contenedores generados por F, para después mostrar que cada uno de estos tamaños es un elemento de M. Puesto que no conocemos la forma particular de F, necesitamos definir un procedimiento general para obtener todos los tamaños posibles. La idea es sencilla y consiste en generar el multiconjunto C de todas las estructuras de todos los contenedores de $F\ Nat$ que tenga al número 1 como su único elemento. Así, si sumamos los elementos de cada estructura en C, obtendremos el tamaño de dicha estructura. Por ejemplo: Si $F\ a = List\ a$, entonces

$$C \quad = \{\!\{\texttt{[ ], [1], [1 ,1], [1, 1, 1],...}\}\!\}$$

y si sumamos los elementos en cada estructura obtenemos el multiconjunto:

$$M \ = \ \ \{\!\{\texttt{0, 1, 2, 3,...}\}\!\}$$

El multiconjunto C es el generado por la función fan definida en la sección 3.2. A continuación formalizamos esta idea.

Definición 3.2 *Sean F un funtor y C el multiconjunto de contenedores de tipo $F\ Nat$ que sólo guardan unos, es decir:*

$$C = \{\!\{\ t \in F\ Nat \mid t \text{ sólo contiene } 1\}\!\}$$

Definimos el multiconjunto de los posibles tamaños de F, denotado $sizes\langle F\rangle$, como sigue:

$$sizes\langle F\rangle\ = \{\!\{\ n \in \mathbb{N}\ b|\ \exists\ t \in C,\ sum\langle F\rangle\ t = n\}\!\}$$

La definición anterior da un procedimiento para construir el multiconjunto $sizes\langle F\rangle$ puesto que $C = fan\langle F\rangle 1$, donde fan es la función politípica, discutida en la sección 3.2. De donde se obtiene la siguiente igualdad:

$$sizes\langle F\rangle\ = \{\!\{\ sum\ \langle F\rangle\ t \mid t \in fan\langle F\rangle\ 1\ \}\!\}$$

La función $sizes\langle F\rangle$ puede calcularse recursivamente, como lo aseguran los siguientes lemas.

Lema 3.1 *Se cumplen las siguientes igualdades:*

$$\begin{array}{rcll} sizes\langle KVoid\rangle & = & \emptyset & (1)\\ sizes\langle KUnit\rangle & = & \{\!\{0\}\!\} & (2)\\ sizes\langle Id\rangle & = & \{\!\{1\}\!\} & (3)\\ sizes\langle F_1 \mid F_2\rangle & = & sizes\langle F_1\rangle \uplus\ sizes\langle F_2\rangle & (4)\\ sizes\langle F_1 \times F_2\rangle & = & sizes\langle F_1\rangle +\ sizes\langle F_2\rangle & (5) \end{array}$$

Demostración. Desarrollamos caso por caso directamente.

Caso 1: $sizes\langle KVoid\rangle = \emptyset$

$$
\begin{aligned}
& sizes\langle KVoid\rangle \\
= \\
& \{\!\{sum\langle KVoid\rangle\ t \mid t \in fan\langle KVoid\rangle 1\}\!\} \\
= \\
& \{\!\{sum\langle KVoid\rangle\ t \mid t \in \emptyset\}\!\} \\
= \\
& \{\!\{sum\langle KVoid\rangle\ \emptyset\}\!\} \\
= \\
& \emptyset
\end{aligned}
$$

Caso 2: $sizes\langle KUnit\rangle = \{\!\{0\}\!\}$

$$
\begin{aligned}
& sizes\langle KUnit\rangle \\
= \\
& \{\!\{sum\langle KUnit\rangle\ t \mid t \in fan\langle KUnit\rangle 1\}\!\} \\
= \\
& \{\!\{sum\langle KUnit\rangle\ t \mid t \in \{\!\{()\}\!\}\}\!\} \\
= \\
& \{\!\{sum\langle KUnit\rangle\ ()\}\!\} \\
= \\
& \{\!\{0\}\!\}
\end{aligned}
$$

Caso 3: $sizes\langle Id\rangle = \{\!\{1\}\!\}$

$$
\begin{aligned}
& sizes\langle Id\rangle \\
= \\
& \{\!\{sum\langle Id\rangle\ t \mid t \in fan\langle Id\rangle 1\}\!\} \\
= \\
& \{\!\{sum\langle Id\rangle\ t \mid t \in \{\!\{1\}\!\}\}\!\} \\
= \\
& \{\!\{\{\!\{sum\langle Id\rangle\ 1\}\!\} \\
= \\
& \{\!\{1\}\!\}
\end{aligned}
$$

Caso 4: $sizes\langle F1 \mid F2\rangle \;=\; sizes\langle F1\rangle \uplus\; sizes\langle F2\rangle$

$$
\begin{array}{l}
sizes\langle F1 \mid F2\rangle \\
= \\
\{\!\{sum\langle F1 \mid F2\rangle\ t \mid t \in fan\langle F1 \mid F2\rangle\ 1\}\!\} \\
= \\
\{\!\{sum\langle F1 \mid F2\rangle\ t \mid t \in \{\!\{Left\ x1 \mid x1 \in fan\langle F1\rangle\ 1\}\!\} \uplus \\
\qquad\qquad\qquad\qquad \{\!\{Right\ x2 \mid x2 \in fan\langle F2\rangle\ 1\}\!\}\} \\
= \\
\{\!\{sum\langle F1 \mid F2\rangle\ t \mid t \in \{\!\{Left\ x1 \mid x1 \in fan\langle F1\rangle\ 1\}\!\}\}\!\} \uplus \\
\{\!\{sum\langle F1 \mid F2\rangle\ t \mid t \in \{\!\{Right\ x2 \mid x2 \in fan\langle F2\rangle\ 1\}\!\}\}\!\} \\
\\
= \ (por\ definición\ de\ sum\langle F1 \mid F2\rangle t) \\
\\
\{\!\{sum\langle F1\rangle\ x1 \mid x1 \in fan\langle F1\rangle\ 1\}\!\} \uplus \\
\{\!\{sum\langle F2\rangle\ x1 \mid x1 \in fan\langle F2\rangle\ 1\}\!\} \\
= \\
sizes\langle F1\rangle \uplus\ sizes\langle F2\rangle
\end{array}
$$

Caso 5: $sizes\langle F1 \times F2\rangle \;=\; sizes\langle F1\rangle + sizes\langle F2\rangle$

$$
\begin{array}{l}
sizes\langle F1 \times F2\rangle \\
= \\
\{\!\{sum\langle F1 \times F2\rangle\ a \mid a \in fan\langle F1 \times F2\rangle\ 1\}\!\} \\
\\
= \ (por\ definición\ de\ fan\langle F1 \times F2\rangle\ 1) \\
\\
\{\!\{sum\langle F1 \times F2\rangle\ (x1, x2) \mid x1 \in fan\langle F1\rangle\ 1, \\
\qquad\qquad\qquad\qquad x2 \in fan\langle F2\rangle\ 1\}\!\} \\
= (por\ definición\ de\ sum\langle F1 \times F2\rangle\ (x1, x2)) \\
\\
\{\!\{sum\langle F1\rangle\ x1 + sum\langle F2\rangle\ x2 \mid \\
\qquad\qquad x1 \in fan\langle F1\rangle\ 1, x2 \in fan\langle F2\rangle\ 1\}\!\} \\
= \\
\{\!\{sum\langle F1\rangle\ x1 \mid x1 \in fan\langle F1\rangle\ 1\}\!\} + \\
\{\!\{sum\langle F2\rangle\ x2 \mid x2 \in fan\langle F2\rangle\ 1\}\!\} \\
= \\
sizes\langle F1\rangle + \ sizes\langle F2\rangle
\end{array}
$$

$\dashv$

El caso de la composición lo mostramos en el siguiente lema:

Lema 3.2 *Si el funtor F involucra solo composiciones admisibles, es decir, composiciones de la forma $H \cdot Id^k$ entonces:*

$$sizes\ \langle F \cdot Id^n \rangle\ = sizes\langle F \rangle *\ \{\!\{n\}\!\}$$

Demostración. La demostración es por inducción sobre F.

Sea $G = Id^n$,

Caso 1: $F = KVoid$

$$\begin{array}{l} sizes\langle KVoid \cdot G \rangle \\ = \\ sizes\langle KVoid \rangle \\ \\ =\ (por\ el\ lema\ 3.1\ ,\ inciso\ 1) \\ \\ \emptyset \\ = \\ \emptyset * \{\!\{n\}\!\} \\ = \\ sizes\langle KVoid \rangle *\ \{\!\{n\}\!\} \end{array}$$

Caso 2: $F = KUnit$

$$\begin{array}{l} sizes\langle KUnit \cdot G \rangle \\ = \\ sizes\langle KUnit \rangle \\ \\ =\ (por\ el\ lema\ 3.1\ ,\ inciso\ 2) \\ \\ \{\!\{0\}\!\} \\ \\ =\ (porque\ \{\!\{n\}\!\}\ es\ simple) \\ \\ \{\!\{0\}\!\} * \{\!\{n\}\!\} \\ = \\ sizes\langle KUnit \rangle *\ \{\!\{n\}\!\} \end{array}$$

Caso 3: $F = Id$

$$
\begin{aligned}
& sizes\langle Id \cdot G\rangle \\
&= \\
& sizes\langle Id\rangle \\
& \\
&= \ (por\ lema\ 3.1\ ,\ inciso\ 3) \\
& \\
& \{\!\{1\}\!\} \\
&= \\
& \{\!\{1\}\!\} * \{\!\{n\}\!\} \\
&= \\
& sizes\langle Id\rangle * \{\!\{n\}\!\}
\end{aligned}
$$

Caso 4: $F = F1 \mid F2$

$$
\begin{aligned}
& sizes\langle (F1 \mid F2) \cdot G\rangle \\
&= \\
& sizes\langle F1 \cdot G \mid F2 \cdot G\rangle \\
& \\
&= \ (por\ el\ lema\ 3.1\ ,\ inciso\ 4) \\
& \\
& sizes\langle F1 \cdot G\rangle \uplus \ sizes\langle F2 \cdot G\rangle \\
& \\
&= \ (por\ H.I) \\
& \\
& (sizes\langle F1\rangle * \ \{\!\{n\}\!\}) \uplus (sizes\langle F2\rangle * \ \{\!\{n\}\!\}) \\
&= \\
& (sizes\langle F1\rangle \uplus \ sizes\langle F2\rangle) * \ \{\!\{n\}\!\} \\
&= \\
& sizes\langle F1|F2\rangle * \ \{\!\{n\}\!\}
\end{aligned}
$$

Caso 5: $F = F1 \times F2$

$$
\begin{aligned}
& sizes\langle (F1 \times F2) \cdot G\rangle \\
&= \\
& sizes\langle F1 \cdot G \times F2 \cdot G\rangle \\
& \\
&= \ (por\ el\ lema\ 3.1\ ,\ inciso\ 5)
\end{aligned}
$$

$$sizes\langle F1 \cdot G\rangle + sizes\langle F2 \cdot G\rangle$$

$$= \ (por\ H.I)$$

$$(sizes\langle F1\rangle * \ \{\!\{n\}\!\}) + (sizes\langle F2\rangle * \ \{\!\{n\}\!\})$$
$$=$$
$$(sizes\langle F1\rangle + \ sizes\langle F2\rangle) * \{\!\{n\}\!\}$$
$$=$$
$$sizes\langle F1 \times F2\rangle * \ \{\!\{n\}\!\}$$

⊣

Ya estamos listos para mostrar la equivalencia entre multiconjuntos y funtores.

Teorema 3.1 *Si el funtor F corresponde al multiconjunto M y si la definición de M solo involucra productos admisibles, entonces*

$$M = sizes\langle F\rangle,$$

es decir, el multiconjunto M tiene como elementos exactamente a los posibles tamaños de la estructura F.

Demostración. La demostración es por inducción sobre M

Caso 1: $M = \emptyset, F = KVoid$

$$\begin{aligned} & sizes\langle F\rangle \\ = & \\ & sizes\langle KVoid\rangle \\ = & \\ & \emptyset \\ = & \\ & M \end{aligned}$$

Caso 2: $M = \{\!\{0\}\!\}, F = KUnit$

$$\begin{aligned} & sizes\langle F\rangle \\ = & \\ & sizes\langle KUnit\rangle \\ = & \\ & \{\!\{0\}\!\} \\ = & \\ & M \end{aligned}$$

Caso 3: $M = M1 + M2, F = F1 \times F2$

$$\begin{array}{rl} & sizes\langle F1 \times F2\rangle \\ = & \\ & sizes\langle F1\rangle + \ sizes\langle F2\rangle \\ = & (por\ H.I.) \\ & M1 + M2 \\ = & \\ & M \end{array}$$

Caso 4: $M = M1 * \{\!\{n\}\!\}, F = F1 \cdot Id^n$

$$\begin{array}{rl} & sizes\langle F1 \cdot Id^n\rangle \\ = & \\ & sizes\langle F1\rangle * \ \{\!\{n\}\!\} \\ = & (por\ H.I.) \\ & M1 * \{\!\{n\}\!\} \\ = & \\ & M \end{array}$$

Con esto queda probado que $M = sizes\langle F\rangle$ para todos los casos de la estructura de tipos.

⊣

Es importante observar que el teorema restringe la definición de M a utilizar solo productos admisibles, es decir, si la expresión M_1 * M_2 aparece en la definición de M entonces M_2 debe ser un multiconjunto simple. Esto se debe a que en otro caso la igualdad no se cumple y por lo tanto la restricción estructural dada en la especificación tampoco. Por ejemplo, supongamos que queremos obtener una estructura F tal que los tamaños de sus contenedores sean las potencias de 3, es decir, tal que $sizes\langle F\rangle$ = {{ 1,3,9,27,... }}. Este multiconjunto puede definirse mediante está ecuación:

$$\text{p3} = \{\!\{\ 1\ \}\!\} \uplus \{\!\{\ 3\ \}\!\} * \text{p3}$$

Como veremos en el siguiente capítulo está ecuación se puede trasformar en un tipo de datos de la siguiente manera:

- La ecuación de multiconjuntos se transforma en la siguiente ecuación funtorial:

$$\texttt{P3} = \texttt{Id} \mid (\texttt{Id} \times \texttt{Id} \times \texttt{Id}) \cdot \texttt{P3}$$

- Al aplicarle un tipo arbitrario a a la ecuación se obtiene:

$$\texttt{P3 a} = \texttt{a} \mid (\texttt{P3 a} \times \texttt{P3 a} \times \texttt{P3 a})$$

- Finalmente llegamos a la siguiente definición en HASKELL:

```
data P3 a = U a | T (P3 a, P3 a, P3 a)
```

Pero está estructura no respeta la restricción deseada. Por ejemplo, es posible almacenar tres elementos como sigue: T (U 1, U 2, U 3). Pero también podemos almacenar cinco elementos de la siguiente forma: T((T(U 1, U 2, U 3)), U 4, U 5). De aquí se observa que la definición de este tipo de datos no respeta el invariante, es decir el hecho de que los contenedores en P3 solo almacenen 3^n elementos.

Esto sucede porque en la definición del multiconjunto p3, el producto {{ 3 }} * p3 no es admisible, porque p3 no es un multiconjunto simple.

Vemos ahora que sucede en el caso contrario, es decir, cuando el factor derecho del producto es admisible: p3 = {{ 1 }} $\uplus$ p3 * {{ 3 }}

La transformación en una ecuación funtorial de la anterior ecuación de multiconjuntos es la siguiente:

$$\texttt{P3} = \texttt{Id} \mid \texttt{P3} \cdot (\texttt{Id} \times \texttt{Id} \times \texttt{Id})$$

A través de la aplicación de la ecuación P3, es posible obtener el tipo de dato en HASKELL:

$$\begin{aligned}\texttt{P3 a} &= \texttt{Id a} \mid \texttt{P3} \cdot (\texttt{Id a} \times \texttt{Id a} \times \texttt{Id a}) \\ &= \texttt{a} \mid \texttt{P3} \cdot (\texttt{a} \times \texttt{a} \times \texttt{a}) \\ &= \texttt{a} \mid \texttt{P3 (a} \times \texttt{a} \times \texttt{a)}\end{aligned}$$

entonces la definición en HASKELL queda como sigue:

```
data P3 a = U a | T P3(a, a, a)
```

Este proceso de manufactura de tipos de datos se explicará a detalle en el siguiente capítulo. Por lo pronto observemos que en este caso, no es posible almacenar cinco elementos, ya que la definción de P3 solo permite agregar elementos mediante una terna, por lo que resulta imposible guardar un número de elementos que no sea una potencia de 3.

En este capítulo hemos desarrollado el concepto de funtor como tipo contenedor y hemos enunciado y demostrado una correspondencia precisa entre multiconjuntos y funtores, la cual sustenta el proceso de manufactura de tipos de datos que describiremos en el siguiente capítulo.

Capítulo 4

Manufactura de tipos de datos

El propósito de este capítulo final, es utilizar toda la teoría desarrollada en los capítulos anteriores, para definir un proceso de manufactura de tipos de datos a partir de una especificación particular. Este proceso se sirve ampliamente del teorema 3.1, y de la captura de restricciones de tamaño mediante ecuaciones de multiconjuntos. Por ejemplo, para construir una estructura cuyos contenedores solo permitan almacenar un número par de elementos, bastará con definir el multiconjunto de números pares mediante una ecuación, y utilizar la correspondencia de multiconjuntos a funtores, obteniendo una ecuación funtorial, la cual se puede transformar fácilmente a un tipo en Haskell.

El proceso de manufactura es el siguiente:

Dada una especificación de algún tipo de dato D, construimos su implementación, a través del siguiente proceso:

1. Definir un sistema de ecuaciones de multiconjuntos S_D el cual utilice solo productos admisibles, de tal manera que se capturen las restricciones de tamaño y estructura dadas en la especificación para los elementos de D.

2. Transformar S_D a un sistema de ecuaciones de funtores F_D, de acuerdo a la correspondencia dada en el capítulo 3 (véase la definición 3.1).

3. Aplicar un tipo arbitrario a a las ecuaciones de F_D, para obtener un sistema de ecuaciones de tipos T_D en forma aplicativa.

4. A partir del sistema de ecuaciones T_D se obtiene un tipo de dato en HASKELL, introduciendo nombres adecuados de constructores en dicha ecuación y siguiendo algunas convenciones propias del lenguaje. Específicamente, la ecuación:

$$F\ a = F_1\ a \mid \ldots \mid F_k\ a$$

Se transforma en la siguiente declaración *data* de HASKELL

```
data F a = C_1 (F_1 a) |...| C_k (F_k a)
```

Esta ecuación implementa al tipo de dato D, aunque los nombres del constructor de tipo F y de los constructores C_i, pueden sustituirse de manera conveniente para dar mayor claridad a la definición. A continuación mostramos cuatro ejemplos sencillos.

Ejemplo 4.1 Se requiere un tipo de dato D, tal que para cualquier número natural n, D tenga un único contenedor de tamaño n.

1. Las restricciones en la especificación corresponden al multiconjunto de números naturales nat = $\{\!\{0,1,2,\ldots\}\!\}$. El sistema de ecuaciones
S_D es entonces:

$$\text{nat} = 0 \uplus (1 + \text{nat})$$

2. La transformación de S_D a un sistema de ecuaciones de funtores F_D es:

$$F_{nat} = \text{KUnit} \mid \text{Id} \times F_{nat}$$

3. El proceso de aplicar un tipo arbitrario a, a la ecuación F_D es el siguiente:

$$\begin{aligned} F_{nat}\ \text{a} &= (\text{KUnit} \mid \text{Id} \times F_{nat})\ \text{a} \\ &= \text{KUnit a} \mid \text{Id a} \times F_{nat}\ \text{a} \\ &= \text{Unit} \mid \text{a} \times F_{nat}\ \text{a} \end{aligned}$$

por lo que el sistema de ecuaciones de tipo es:

$$F_{nat}\ \text{a} = \text{Unit} \mid \text{a} \times F_{nat}\ \text{a}$$

$$\text{data}\ F_{nat}\ \text{a} = C_1\ \text{Unit} \mid C_2\ (\text{a} \times F_{nat}\ \text{a})$$

4. Finalmente utilizando nombres comunes para el tipo y los constructores, así como ciertas convenciones de HASKELL, en este caso la omisión del tipo unitario en el primer constructor y la preferencia por usar una función de orden superior [1] como segundo constructor, llegamos a la siguiente definición.

```
data List a = Nil | Cons a (List a)
```

Por lo tanto nuestra especificación original del tipo D corresponde al tipo de dato de listas finitas.

Ejemplo 4.2 Se requiere un tipo de dato D que contenga una infinidad de contenedores vacíos.

1. Las restricciones en la especificación corresponden al multiconjunto: ceros = $\{\!\{0,0,0,\ldots\}\!\}$. Por lo tanto, el sistema de ecuaciones de multiconjuntos S_D queda como sigue:

$$\text{cero} = 0 \uplus \text{cero}$$

2. La transformación de S_D a un sistema de ecuaciones de funtores F_{cero} es:

$$F_{cero} = \text{KUnit} \mid F_{cero}$$

3. El proceso de aplicar un tipo arbitrario a, a la ecuación F_{cero} es el siguiente:

$$\begin{aligned} F_{cero}\ \text{a} &= (\text{KUnit} \mid F_{cero})\ \text{a} \\ &= \text{KUnit a} \mid F_{cero}\ \text{a} \\ &= \text{Unit} \mid F_{cero}\ \text{a} \end{aligned}$$

entonces T_D es:

$$F_{cero}\ \text{a} = \text{Unit} \mid F_{cero}\ \text{a}$$

4. La transformación de T_D a una definición *data* en HASKELL queda como sigue:

[1] Con esto nos referimos a que el constructor `Cons` sea de tipo `Cons:a -> List a -> List a` y no `Cons:(a, List a)-> List a`

```
data Nat a = Zero | Succ (Nat a)
```

Ejemplo 4.3 Se requiere un tipo de dato D tal que para cualquier número natural n, D tenga un único contenedor de tamaño 2n .

1. Las restricciones en la especificación corresponden al multiconjunto de número pares, es decir, al multiconjunto pares = $\{\!\{0,2,4, \ldots\}\!\}$, por lo que la definición del sistema de ecuaciones de muliconjuntos S_D es:

$$\text{pares} = 0 \uplus (\text{pares} + 2)$$

2. La transformación de S_D a un sistema de ecuaciones de funtores F_{pares} es:

$$F_{pares} = KUnit \mid F_{pares} \times (Id \times Id)$$

3. El proceso de aplicar un tipo arbitrario a, a la ecuación F_{pares} es el siguiente:

$$\begin{aligned} F_{pares}\ \text{a} &= (\text{KUnit} \mid F_{pares} \times (Id \times Id))\ a \\ &= \text{KUnit a} \mid (F_{pares} \times (Id \times Id))\ a \\ &= \text{Unit} \mid F_{pares}\ a \times (Id\ a \times Id\ a) \\ &= \text{Unit} \mid F_{pares}\ a \times (a \times a) \end{aligned}$$

entonces T_D:

$$F_{pares}\ \text{a} = \text{Unit} \mid F_{pares}\ a \times (a \times a)$$

4. La transformación de T_D a una definición *data* en HASKELL, es la siguiente:

```
data Even a = NilE | ConsE (Even a) a a
```

Aquí el constructor $NilE$ corresponde a la lista vacía, mientras que el constructor $ConsE$ corresponde a la operación de agregar dos elementos a la cola de una lista dada. Como ésta es la única forma de agregar elementos, claramente se cumple que la longitud de una lista siempre es par. Algunos ejemplos son:

(a) `ConsE (NilE) 0 1`

(b) `ConsE (ConsE (NilE) 0 1) 2 3`

(c) `ConsE (ConsE (ConsE (NilE) 0 1) 2 3) 4 5`

Ejemplo 4.4 Se requiere un tipo de dato D tal que para cualquier número natural n, D tenga un único contenedor de tamaño 3^n. El multiconjunto correspondiente es:

$$\text{p3} = \{\!\{\ 1, 3, 9, 27, 83, \ldots \}\!\}$$

1. La definición del sistema de ecuaciones de multiconjuntos S_D:

$$\text{p3} = \{\!\{\ 1 \}\!\} \uplus (\ \text{p3} * \{\!\{\ 3 \}\!\})$$

2. La transformación de S_D a un sistema de ecuaciones de funtores F_{p3} es:

$$F_{p3} = \text{Id} \mid F_{p3} \cdot (\ \text{Id} \times \text{Id} \times \text{Id})$$

3. El proceso de aplicar un tipo arbitrario a, a la ecuación F_{p3} es el siguiente:

$$\begin{aligned} F_{p3}\ \text{a} &= \text{Id a} \mid F_{p3} \cdot (\text{Id a} \times \text{Id a} \times \text{Id a}) \\ &= \text{a} \mid F_{p3} \cdot (\text{a} \times \text{a} \times \text{a}) \\ &= \text{a} \mid F_{p3}\ (\text{a} \times \text{a} \times \text{a}) \\ \text{entonces } T_D &= F_{p3}\ \text{a} = \text{a} \mid F_{p3}\ (\text{a} \times \text{a} \times \text{a}) \end{aligned}$$

4. La transformación de T_D a una definición *data* en Haskell queda como sigue:

```
data P3 a = U a | T (P3 (a, a, a))
```

Es importante observar que el tipo $P3\ a$ es un ejemplo de tipo anidado o heterogéneo, debido a que en la llamada recursiva el parámetro cambia de a a (a, a, a). Esta clase de definiciones de tipo en realidad genera familias infinitas de tipos parametrizados. A diferencia de un tipo homogéneo o no anidado, como las listas comunes, es imposible aislar a un elemento de esta familia. Para programar con esta clase de tipos requerimos de técnicas avanzadas de la programación funcional, tales como la recursión polimórfica. Algunos ejemplos son:

(a) `U(1)`

(b) `T(U(1,2,3))`

(c) `T(U((1,2,3),(4,5,6),(7,8,9)))`

La idea de esta definición es construir ternas, ternas de ternas, etc, por lo que el constructor T puede pensarse como un constructor de ternas, a partir de otras encapsuladas previamente por el constructor U.

A continuación presentamos algunos ejemplos más elaborados.

4.1. Matrices de Toeplitz

Las matrices de Toeplitz (denominadas así en honor a Otto Toeplitz), son una clase especial de matrices de importancia en algunas áreas de las matemáticas. Una matriz de Toeplitz es de la forma $M = (a_{i,j})$ de dimensión $n \times n$ y que debe cumplir la siguiente restricción estructural:

$$a_{i,j} = a_{i-1,j-1} \; para \; 1 < i, j \leq n.$$

Esta restricción formal corresponde al hecho de que en una matriz de Toeplitz toda diagonal descendiente de izquierda a derecha, es decir toda diagonal paralela a la diagonal principal, debe ser constante.
Por ejemplo, cuando $n = 5$, la matriz de Toeplitz presenta la siguiente estructura:

$$M = \begin{pmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{pmatrix}$$

Se observa que debido a la restricción estructural, solo es necesario conocer los elementos del primer renglón y de la primera columna. Luego entonces, dada cualquier n $\in \mathbb{N}$ para representar a una matriz de Toeplitz de tamaño $(n + 1) \times (n + 1)$, es suficiente conocer $2n + 1$ elementos. De manera que para implementar un tipo de dato que represente matrices de Toeplitz, basta buscar un tipo que incluya un contenedor de tamaño $2n + 1$ para cualquier n. Es decir, el multiconjunto de los tamaños posibles para esta estructura es el multiconjunto de todos los números impares: $odd = \{\!\{1,3,5,7,\ldots\}\!\}$.
El proceso de manufactura es el siguiente:

1. Las restricciones en la especificación corresponden a la siguiente definición del sistema de ecuaciones de multiconjuntos S_D:

$$impar = 1 \uplus (impar + 2)$$

2. La transformación de S_D a un sistema de ecuaciones de funtores F_I es:

$$F_I = Id \mid F_I \times (Id \times Id)$$

3. El proceso de aplicar un tipo arbitrario a, a la ecuación F_I es el siguiente:

$$\begin{aligned} F_I\ a &= Id\ a | F_I\ a \times (Id \times Id)\ a \\ &= Id\ a | F_I\ a \times (Id\ a \times Id\ a) \\ &= a\ | F_I\ a \times (a \times a) \\ \text{entonces } T_D = F_I\ a &= a \mid F_I\ a \times (a \times a) \end{aligned}$$

4. La transformación de T_D a una definición *data* en HASKELL:

```
data Toeplitz a = Corner a | Extend (Toeplitz a) a a
```

El elemento x que se encuentra en la esquina superior izquierda de una matriz M está representado por *Corner* x, mientras que la operación *Extend* $m\ r\ c$ corresponde a extender M, agregando el elemento r al primer renglón y el elemento c a la primera columna (los elementos restantes se obtienen automáticamente de la restricción estructural). El siguiente ejemplo es la representación de la matriz M, de dimensión 5×5:

`Extend (Extend (Extend (Extend (Corner` a_{11} `)` $a_{12}\ a_{21}$ `)` $a_{13}\ a_{31}$ `)` $a_{14}\ a_{41}$ `)` $a_{15}\ a_{51}$

4.2. Matrices Cuadradas

Las matrices cuadradas, son matrices $M=(a_{i,j})$ en donde el número de renglones es igual al número de columnas, es decir, son de dimensión $n \times n$. Por ejemplo:

$$M = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

Por la restricción estructural, cualquier matriz cuadrada contiene n^2 elementos, por lo tanto, para obtener una representación de dichas matrices es

necesario crear un tipo que tenga un único contenedor de tamaño justamente n^2.

A continuación, mostramos el desarrollo del proceso de manufactura:

1. Las restricciones en la especificación corresponden al multiconjunto de números cuadrados, es decir $square = \{\!\{0,1,2,4,9,16,25, \ldots\}\!\}$. De lo anterior, el sistema de ecuaciones de multiconjuntos S_D corresponde a:

$$\begin{aligned} square &= squareFrom\ 0 \\ squareFrom\ n &= n * n \uplus squareFrom\ (1+n) \end{aligned}$$

La idea de este sistema de ecuaciones consiste en generar recursivamente todos los números cuadrados a partir de un número n dado, por lo que el multiconjunto de todos los números cuadrados es igual al multiconjunto de los números cuadrados generados a partir del cero.

2. La transformación de S_D a un sistema de ecuaciones de funtores F_c es:

$$\begin{aligned} \texttt{Square} &= \texttt{SquareFrom}\ (\texttt{KUnit}) \\ \texttt{SquareFrom m} &= \texttt{m} \cdot \texttt{m} \mid \texttt{SquareFrom}\ (\texttt{Id} \times \texttt{m}) \end{aligned}$$

3. El proceso de aplicar un tipo arbitrario a, a la ecuación F_c es el siguiente:

$$\texttt{Square a} = \texttt{SquareFrom}\ (\texttt{KUnit})\ \texttt{a}$$

$$\begin{aligned} \texttt{SquareFrom m a} &= (\texttt{m} \cdot \texttt{m})\ \texttt{a} \mid \texttt{SquareFrom}\ (\texttt{Id} \times \texttt{m})\ \texttt{a} \\ &= \texttt{m(m a)} \mid \texttt{SquareFrom}\ (\texttt{Id} \times \texttt{m})\ \texttt{a} \end{aligned}$$

entonces T_D es:

$$\begin{aligned} \texttt{Square a} &= \texttt{SquareFrom}\ (\texttt{KUnit})\ \texttt{a} \\ \texttt{SquareFrom m a} &= \texttt{m(m a)} \mid \texttt{SquareFrom}\ (\texttt{Id} \times \texttt{m})\ \texttt{a} \end{aligned}$$

4. La transformación de T_D a una definición *data* en HASKELL, es la siguiente:

$$\begin{aligned} \texttt{type SquareM a} &= \texttt{SquareMfrom NilC a} \\ \texttt{data SquareMfrom m a} &= \texttt{Zero(m(m a))} \mid \texttt{Succ(SquareMfrom (ConsC m) a)} \\ \texttt{data NilC a} &= \texttt{NilM} \\ \texttt{data ConsC m a} &= \texttt{Cons a (m a)} \end{aligned}$$

Aquí el tipo $ConsC\ m\ a$, corresponde a $Id \times m$ y m corresponde a un constructor arbitrario de matrices que en el caso inicial es el constructor de

matrices vacías NilC.

Observemos que el tipo *SquareMfrom* es nuevamente un tipo anidado o heterogéneo, debido a que en la llamada recursiva el primer parámetro cambia de m a $ConsC\ m$. Sin embargo, este tipo es más complicado que el del ejemplo de potencias de tres, puesto que aquí el parámetro que cambia m no es un tipo, sino un constructor de tipos. Esta clase de definiciones se conoce como tipos anidados de orden superior.

La idea central de la representación anterior corresponde a la implementación común para representar matrices cuadradas, en donde se utilizan listas que contengan como elementos listas para almacenar cada elemento de la matriz. Aunque la diferencia que existe en este caso, es que la definición misma del tipo asegura que se cumpla la restricción de tamaño, es decir, obliga a que el número de renglones y columnas sea el mismo, además de utilizar los constructores $Zero$ y $Succ$ para representar el tamaño (n) de un renglón o columna de la matriz.

Veamos la representación para matrices cuadradas de dimensión 1 y 2 respectivamente:

(a)
```
Succ(Zero(Cons(Cons 1 NilM) (NilM)))
```

(b)
```
Succ(Succ(Zero(Cons(Cons 1(Cons 2 NilM))
               (Cons (Cons 3(Cons 4 NilM)) (NilM)))))
```

4.3. Árboles de Braun

Los árboles de Braun son árboles binarios balanceados, posiblemente vacíos, que cumplen la siguiente propiedad: para cualquier nodo, el subárbol izquierdo a partir de ese nodo, tiene exactamente el mismo número de elementos que el subárbol derecho, o uno más. Estos árboles son útiles para implementar ciertas estructuras como arreglos flexibles y colas de prioridad. No existe restricción de tamaño y la restricción estructural implica las siguientes condiciones:

- Si el árbol de Braun T tienen tamaño $2n + 1$, entonces los subárboles izquierdo y derecho son de tamaño n
- Si el árbol de Braun T tiene tamaño $2n + 2$, entonces el subárbol izquierdo tiene tamaño $n + 1$ y el subárbol derecho tiene tamaño n

En la figura 4.3 se muestran los árboles de Braun de tamaño 1 a 7.

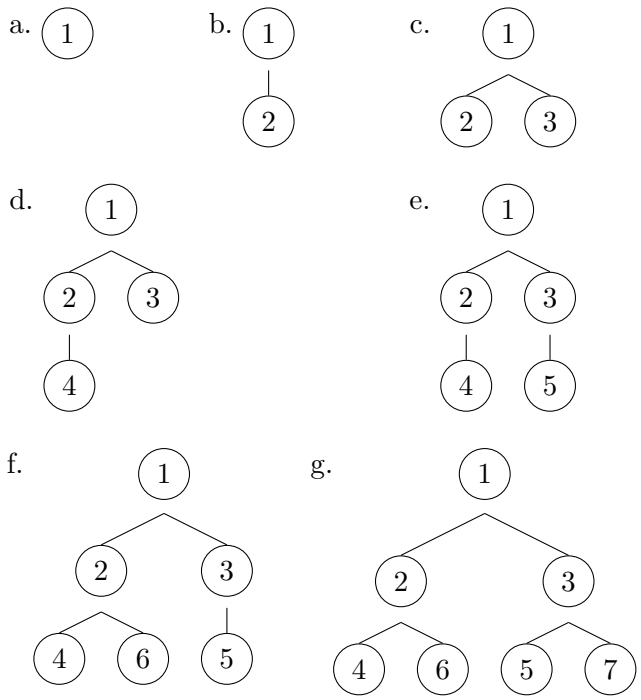

Figura 4.3: Árboles de Braun

Según las restricciones en la especificación de un árbol de Braun, estos pueden tener cualquier tamaño $n \in \mathbb{N}$, pero de acuerdo a los ejemplos anteriores resulta conveniente representar a cualquier natural mediante su paridad binaria, dejando residuo 1 o 2, es decir considerando números de la forma $(2n + 1)$ o $(2n + 2)$. De esto se puede obtener una representación de los números naturales positivos, es decir, de los posibles tamaños de un árbol de Braun no vacío, a tráves del sistema binario con dígitos 1 y 2, que será descrito a continuación.

Para representar un número positivo p en el sistema binario con dígitos 1 y 2, procederemos como sigue:

1. Descomponer p en la forma $(2n+1)$ o $(2n+2)$, es decir dividir p entre 2 de tal manera que el residuo sea 1 o 2. El residuo obtenido será el siguiente dígito menos significativo de la representación binaria.

2. Repetir el procedimiento anterior con el valor que se obtuvo de n, hasta que n sea igual a cero.

Por ejemplo, para obtener la representación binaria del número 5 se realiza la siguiente descomposición:

$$5 = 2 * (2) + 1$$

$$2 = 2 * (0) + 2$$

por lo que la descomposición binaria de 5 es 21.

Este proceso de descomposición nos sirve también para construir árboles de Braun de cualquier tamaño, observando que dado un árbol de tamaño n, es posible construir un nuevo árbol de Braun de tamaño $2n+1$, simplemente pegando dos copias del árbol de tamaño n con una nueva raíz. El inconveniente surge al construir árboles de tamaño $2n + 2$, ya que en este caso el árbol de la derecha es de tamaño n y el de la izquierda es de tamaño $n + 1$, pero no es fácil insertar un elemento al árbol de tamaño n de tal forma que se sigan respetando las restricciones de un árbol de Braun. Por lo tanto, la idea es realizar la descomposición nuevamente a partir del tamaño $n+1$. En conclusión la construcción de un árbol de tamaño n requiere de iterar las funciones $One(x) = 2x + 1$ y $Two(x) = 2x + 2$ para distintos valores de x. Veamos un ejemplo:

Para construir un árbol de tamaño 16, la descomposición es:

$$\begin{aligned} 16 &= 2(7) + 2 \\ 7 &= 2(3) + 1 \\ 3 &= 2(1) + 1 \\ 1 &= 2(0) + 1 \end{aligned}$$

lo cual corresponde a la iteración de las funciones One y Two

$$\begin{aligned} 16 &= Two(7) \\ &= Two(One(3)) \\ &= Two(One(One(1))) \\ &= Two(One(One(One(0)))) \end{aligned}$$

Esto significa que para construir el árbol de tamaño 16 necesitamos del árbol de tamaño 7; para la construcción de éste, se requiere del árbol de tamaño

3, el cual necesita el árbol de tamaño 1 que se construye a partir del árbol vacío. Las aplicaciones de la función *One* corresponden, como ya se dijo, a construir un árbol cuyos subárboles izquierdo y derecho son el mismo. El caso de la función Two requiere de una nueva iteración, puesto que el árbol de tamaño 16 debe tener subárboles de tamaño 8 y 7 respectivamente, por lo que debemos calcular la descomposición del número 8.

$$\begin{aligned} 8 &= 2(3)+2 \\ 3 &= 2(1)+1 \\ 1 &= 2(0)+1 \end{aligned}$$

que corresponde a:

$$\begin{aligned} 8 &= Two(3) \\ &= Two(One(1)) \\ &= Two(One(One(0))) \end{aligned}$$

Lo anterior significa que para construir el árbol de tamaño 8 necesitamos del árbol de tamaño 3, el cual requiere la construcción del árbol de tamaño 1 que es construido a partir del árbol vacío. En este caso, al aplicar la función Two se requiere de una iteración extra, ya que el árbol de tamaño 8 necesita los subárboles de tamaño 4 y 3 respectivamente, por lo tanto calculamos la descomposición del número 4.

$$\begin{aligned} 4 &= 2(1)+2 \\ 2 &= 2(0)+1 \end{aligned}$$

la cual corresponde a:

$$\begin{aligned} 4 &= Two(1) \\ &= Two(One(0)) \end{aligned}$$

Esto nos dice que para construir el árbol de tamaño 4 es necesario el árbol de tamaño 1 que, como sabemos, es construido a partir del árbol vacío. Como observamos, se requiere nuevamente de una iteración de la función Two, porque el árbol de tamaño 4 requiere del subárbol de tamaño 2 y 1, por lo cual calculamos la descomposición del número 2.

$$2 = 2(0)+2$$

Que corresponde a:

$$2 = One(0)$$

De este ejemplo se observa, que para obtener los posibles tamaños de un árbol de Braun a partir de n, basta construir los posibles tamaños a partir

de $2n+1$ y $2n+2$. Esta es la idea del sistema de multiconjuntos S_D para árboles de Braun.

$$\begin{aligned} braun &= braunFrom\ 0 \\ braunFrom\ n &= n \uplus braunFrom(1+n+n) \uplus braunFrom(2+n+n) \end{aligned}$$

Ahora, la transformación de S_D al sistema de ecuaciones de funtores F_{braun} y $F_{braunFrom}$ es:

$$\begin{aligned} &\mathtt{F_{braun}} \\ &\quad = \mathtt{F_{braunFrom}\ KUnit} \\ &\mathtt{F_{braunFrom}\ t} \\ &\quad = \mathtt{t} \mid \mathtt{F_{braunFrom}(Id \times t \times t)} \mid \mathtt{F_{braunFrom}(Id \times Id \times t \times t)} \end{aligned}$$

Al aplicar un tipo arbitrario a, a la ecuación F_{braun} y $F_{braunFrom}$ obtenemos lo siguiente:

$$\begin{aligned} &\mathtt{F_{braun}\ a} \\ &\quad = \mathtt{F_{braunFrom}\ Unit} \\ &\mathtt{F_{braunFrom}\ t\ a} \\ &\quad = \mathtt{t\ a} \mid \mathtt{F_{braunFrom}(Id \times t \times t)\ a} \mid \mathtt{F_{braunFrom}(Id \times Id \times t \times t)\ a} \\ &\quad = \mathtt{t\ a} \mid \mathtt{F_{braunFrom}(Id\ a \times t\ a \times t\ a)} \\ &\qquad \mid \mathtt{F_{braunFrom}(Id\ a \times Id\ a \times t\ a \times t\ a)} \\ &\quad = \mathtt{t\ a} \mid \mathtt{F_{braunFrom}(a \times t\ a \times t\ a)} \\ &\qquad \mid \mathtt{F_{braunFrom}(a \times a \times t\ a \times t\ a)} \end{aligned}$$

La transformación de T_D en una definición *data* en HASKELL es:

```
type Empty a = E
type Braun = BraunFrom Empty
data BraunFrom t a = Base (t a)
                   | One (BraunFrom (Bin t t) a)
                   | Two (BraunFrom (Bin (Bin' t) t) a)
```

donde,

```
data Bin' t a  =  Bin' a (t a)
data Bin t1 t2 a =  Bin a (t1 a) (t2 a)
```

Los nombres de los constructores (`Base, One, Two`) tienen el propósito de codificar en la definición misma del árbol, su tamaño expresado en el sistema binario con dígitos 1 y 2. Esta representación también puede verse como la

iteración de las funciones *One* y *Two* a partir de cero, hasta llegar al tamaño del árbol en cuestión[2].

Para dejar más clara la anterior representación, veamos algunos ejemplos (su representación gráfica se muestra en la Figura 4.3):

(a) `One(Base(Bin 1 Unit Unit))`

(b) `Two(Base(Bin 1 (Bin' 2 Unit) Unit))`

(c) `One(One(Base(Bin 1 (Bin 2 Unit Unit) (Bin 3 Unit Unit))))`

(d)
```
One(Two(Base(Bin 1 (Bin' 2 (Bin 4 Unit Unit))
                           (Bin 3 Unit Unit))))
```

(e)
```
Two(One(Base(Bin 1 (Bin 2 (Bin' 4 Unit) Unit)
                   (Bin 3 (Bin' 5 Unit) Unit))))
```

(f)
```
Two(Two(Base(Bin 1 (Bin' 2 (Bin 4 (Bin' 6 Unit) Unit))
                           (Bin 3 (Bin' 5 Unit) Unit))))
```

(g)
```
One(One(One(Base(Bin 1
          (Bin 2 (Bin 4 Unit Unit) (Bin 6 Unit Unit))
          (Bin 3 (Bin 5 Unit Unit)(Bin 7 Unit Unit))))))
```

Cabe mencionar que esta representación es aportación nuestra, y resulta en nuestra opinión, conceptualmente más simple que la implementación original de Hinze. Sin embargo, consideramos adecuado presentar también dicha implementación puesto que corresponde a una representación más explicita de los árboles no balanceados, es decir, los generados por la operación $x \mapsto 2x+2$. En particular, nuestra representación no hace explícitos los árboles vacíos en este caso, situación se resuelve ahora.

El proceso de manufactura es el siguiente:

1. Las restricciones en la especificación corresponden a la siguiente definición del sistema de ecuaciones de multiconjuntos S_D:

$$\begin{aligned} \texttt{braun} &= \texttt{braun}'\ 0\ 1 \\ \texttt{braun}'\ \texttt{n}\ \texttt{n}' &= \texttt{n} \uplus \texttt{braun}'(1+\texttt{n}+\texttt{n})(1+\texttt{n}'+\texttt{n}) \\ &\quad \uplus \texttt{braun}'(1+\texttt{n}'+\texttt{n})(1+\texttt{n}'+\texttt{n}') \end{aligned}$$

[2]obsérvese que en este caso la representación debe leerse al revés, por ejemplo, el árbol de tamaño 5 es de la forma `Two(One(Base t))`, mientras que $5 = One(Two(0))$

A través de la ecuación *braun* se generan los posibles tamaños de árboles de Braun, iniciando con dos semillas, que corresponden a los árboles con tamaños 0 y 1, es decir, el árbol vacío y las hojas. La ecuación *braun'* construye árboles de Braun a partir de dos tamaños arbitrarios n y n' (en realidad para construir árboles de Braun se debe cumplir que $n' = n + 1$, pero la definición debe ser general), siendo la idea de su definición recursiva, nuevamente, la construcción de las iteraciones de las operaciones $x \mapsto 2x+1$ y $x \mapsto 2x+2$.

2. La transformación de S_D a un sistema de ecuaciones de funtores F_{braun} es:

```
     F_braun  = F'_braun (KUnit) Id
F'_braun t t' = t | F'_braun (Id × t × t) (Id × t' × t)
                  | F'_braun (Id × t' × t) (Id × t' × t')
```

3. El proceso de aplicar un tipo arbitrario a, a la ecuación F_{braun} es el siguiente:

```
F_braun a
        = F_braun' (KUnit) Id a
        = F_braun' (KUnit a) Id a
        = F_braun' Unit a

F_braun' t t' a
    = f a |  F_braun' (Id × t × t) (Id × t' × t) a
       | F_braun' (Id × t' × t) (Id × t' × t') a
    = f a | F_braun'  (Id a × t a × t a) (Id a × t' a × t a)
       | F_braun' (Id a × t' a  × t a) (Id a × t' a × t' a)
    = f a | F_braun' (a × t a × t a) (a × t' a × t a)
       | F_braun' (a × t' a × t a) (a × t' a × t' a)
```

4. La transformación de T_D a una definición *data* en HASKELL es:

```
type Empty a = E
type Braun = Braun' (Empty) Id
data Braun' t t' a
          = Base (t a) | One (Braun' (Bin t t) (Bin t' t) a)
                       | Two (Braun' (Bin t' t) (Bin t' t') a)
```

donde `Bin x y` representa el tipo de dato que construye un árbol binario de la manera usual, es decir, une dos árboles (subárbol izquierdo y derecho), no

necesariamente del mismo tipo, con un nuevo nodo raíz. Lo cual corresponde al tipo : a $\times$ t_1 a $\times$ t_2 a. En HASKELL se define el tipo de datos de la siguiente forma:

```
data Bin t1 t2 a = Bin a (t1 a) (t2 a)
```

Veamos nuevamente los ejemplos la Figura 4.1 usando está nueva implementación:

(a) `One(Base(Bin 1 Unit Unit))`

(b) `Two(Base(Bin 1 2 Unit))`

(c) `One(One(Base(Bin 1 (Bin 2 Unit Unit) (Bin 3 Unit Unit))))`

(d) `One(Two(Base(Bin 1 (Bin 2 4 Unit) (Bin 3 Unit Unit))))`

(e) `Two(One(Base(Bin 1 (Bin 2 4 Unit) (Bin 3 5 Unit))))`

(f) `Two(Two(Base(Bin 1 (Bin 2 4 6) (Bin 3 5 Unit))))`

(g) `One(One(One(Base(Bin 1 (Bin 2 4 6) (Bin 3 5 7)))))`

4.4. Árboles híbridos (*Fork-node trees*)

Los árboles híbridos surgen como solución a la siguiente restricción estructural:

Se requiere almacenar en un árbol binario perfecto[3] cualquier número de elementos. Puesto que un árbol binario perfecto de altura n tiene $2^{n+1} - 1$ nodos, para guardar un número que no es de esta forma se deben dejar nodos vacíos o no etiquetados. Para esto damos la siguiente restricción estructural:

1. Para cada nivel, todos o ninguno de sus nodos están etiquetados.

2. Los nodos del último nivel están etiquetados.

Esta estructura es poco conocida y el nombre dado en [3] es *fork-node trees*, debido a que en su definición se utilizan dos constructores para fabricar árboles, los cuales se llaman *Fork* y *Node*. Decidimos en este trabajo darles

[3]Recordemos que un árbol binario perfecto es un árbol binario completo que tiene todos los niveles llenos

el nombre de *híbridos*, debido a que tendrán dos clases de nodos, los etiquetados y los no etiquetados.

Algunos ejemplos de árboles híbridos son:

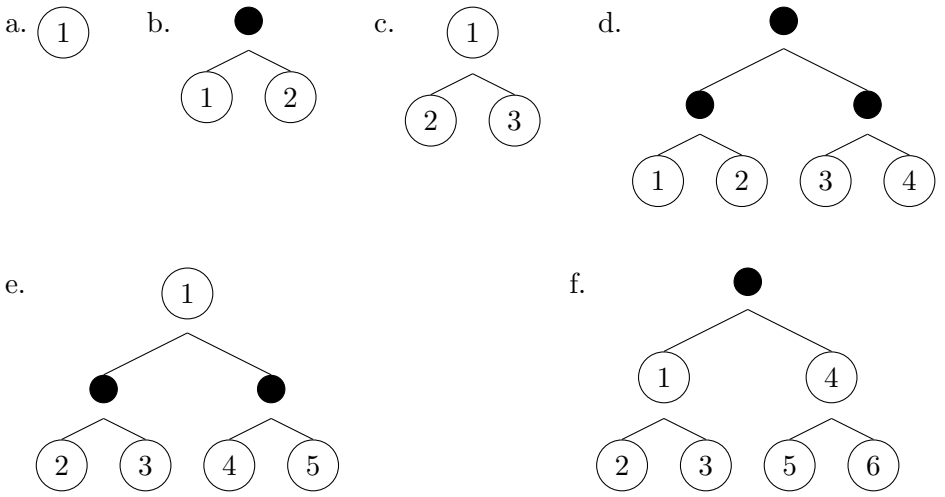

Figura 4.4: Árboles híbridos

Es claro que para guardar k elementos en un árbol perfecto, necesitamos un número n mínimo, tal que $k < 2^{n+1} - 1$. De esta manera los k elementos se guardaran en un árbol perfecto de altura n. Sin embargo, aún nos queda el problema de decidir qué nodos etiquetar. Resolvemos estos problemas nuevamente utilizando la descomposición binaria de un número, aunque en este caso utilizamos el sistema binario común con dígitos 0 y 1.

La idea general es iterar las funciones $Zero(x) = 2x$ y $One(x) = 2x + 1$, para representar la descomposición binaria del número de elementos a guardar. Por ejemplo, si queremos guardar 5 elementos, procedemos como sigue:

$$\begin{aligned} 5 &= 2(2) + 1 \\ 2 &= 2(1) + 0 \\ 1 &= 2(0) + 1 \end{aligned}$$

Por lo que 5 en binario se representa mediante 101, que corresponde tambien a la siguiente iteración de las funciones *One* y *Zero*:

$$\begin{aligned} 5 &= One(2) \\ &= One(Zero(1)) \\ &= One(Zero(One(0))) \end{aligned}$$

A partir de esta iteración, el árbol correspondiente se construye identificando a la expresión $Zero(x)$ con el proceso de construir un árbol perfecto que guarda $2x$ elementos, colgando dos árboles perfectos que guardan x elementos, de un nodo raíz no etiquetado. Similarmente, la expresión $One(x)$ genera un árbol perfecto que guarda $2x + 1$ elementos, al colgar dos árboles que guardan x elementos, de un nodo raíz etiquetado.

Más aún, puesto que el nivel k de un árbol perfecto tiene 2^k nodos, la descomposición binaria nos indica de manera precisa que niveles están vacíos y cuales están llenos. A saber, el k-ésimo nivel está etiquetado si y solo si el k-ésimo dígito de la descomposición binaria es igual a 1.

Finalmente, observamos que si ya contamos con un árbol que guarda n elementos, es posible generar fácilmente a partir de éste, árboles que guardan $2n$ y $2n+1$ elementos. Repitiendo este proceso, podemos construir un árbol que guarda cualquier número de elementos. Está es la idea del sistema de multiconjuntos S_D para árboles $f$ork-node:

$$\begin{aligned} \texttt{nat} &= \texttt{0} \uplus \texttt{pos 1} \\ \texttt{pos n} &= \texttt{n} \uplus \texttt{pos}(\texttt{n} * \texttt{2}) \uplus \texttt{pos}(\texttt{1} + \texttt{n} * \texttt{2}) \\ &= \texttt{n} \uplus \texttt{pos}(\texttt{n} + \texttt{n}) \uplus \texttt{pos}(\texttt{1} + \texttt{n} + \texttt{n}) \end{aligned}$$

Obsérvese que a través de la primera ecuación se generan todos los posibles tamaños (número de nodos etiquetados) de un árbol $f$ork-node, mientras que la segunda ecuación construye los posibles tamaños a partir de n, usando las construcciones descritas por $Zero$ y One.

Ahora, la transformación de S_D a un sistema de ecuaciones de funtores es:

$$\begin{aligned} \texttt{Nat} &= \texttt{KUnit} \mid \texttt{PosId} \\ \texttt{Pos f} &= \texttt{f} \mid \texttt{Pos}(\texttt{f} \times \texttt{f}) \mid \texttt{Pos}(\texttt{Id} \times (\texttt{f} \times \texttt{f})) \end{aligned}$$

A través del siguiente proceso aplicamos un tipo arbitrario a, al sistema de ecuaciones de funtores para obtener T_D:

$$
\begin{array}{rcl}
\texttt{Nat a} & = & \texttt{KUnit a} \mid \texttt{Pos Id a} \\
& = & \texttt{Unit} \mid \texttt{Pos a} \\
\texttt{Pos f a} & = & \texttt{f a} \mid \texttt{Pos (f} \times \texttt{f) a} \mid \texttt{Pos(Id} \times \texttt{(f} \times \texttt{f)) a} \\
& = & \texttt{f a} \mid \texttt{Pos(f a} \times \texttt{f a)} \mid \texttt{Pos(Id a} \times \texttt{(f a} \times \texttt{f a))} \\
& = & \texttt{f a} \mid \texttt{Pos(f a} \times \texttt{f a)} \mid \texttt{Pos(a} \times \texttt{(f a} \times \texttt{f a))}
\end{array}
$$

Así, la transformación de T_D a una definición *data* en HASKELL es:

$$
\begin{array}{rcl}
\texttt{data Fork t a} & = & \texttt{Fork (t a)(t a)} \\
\texttt{data Node t a} & = & \texttt{Node a (t a)(t a)} \\
\texttt{data Vector a} & = & \texttt{Empty} \mid \texttt{NonEmpty(Vector}' \texttt{ Id a)} \\
\texttt{data Vector}' \texttt{ t a} & = & \texttt{Base(t a)} \mid \texttt{Zero(Vector}'\texttt{(Fork t) a)} \\
& & \mid \texttt{One(Vector}'\texttt{(Node t) a)}
\end{array}
$$

Por ejemplo, el vector (1,2,3) tiene tamaño 3, número cuya descomposición binaria es:

$$
\begin{array}{rcl}
3 & = & 2(1) + 1 \\
1 & = & 2(0) + 1
\end{array}
$$

a partir de esto obtenemos la iteración de las funciones *One* y *Zero*:

$$
\begin{array}{rcl}
3 & = & One(1) \\
& = & One(One(1))
\end{array}
$$

Esto quiere decir, que la implementación de un árbol $f$ork-node que almacena 3 elementos, queda como sigue:

```
NonEmpty(One(Base(Node 1 (2) (3))))
```

Notemos que el número de elementos almacenados en el árbol está representado por los constructores *NonEmpty*, *One* y *Zero*. La idea general es sustituir en la descomposición binaria, el dígito 1 por el constructor *NonEmpty* y *One*, y el dígito 0 por el constructor *Zero*, tomando en cuenta que el primer dígito será el bit más significativo.

Para finalizar está sección, veamos los ejemplos de la figura 4.4

(a) `NonEmpty(Base 1)`

(b)
```
NonEmpty(Zero(Base(Fork (1) (2))))
```

(c)
```
NonEmpty(One(Base(Node 1 (2) (3))))
```

(d)
```
NonEmpty(Zero(Zero(Base(Fork (Fork (1) (2))
                             (Fork (3) (4))))))
```

(e)
```
NonEmpty(Zero(One(Base(Node 1 (Fork (2) (3))
                              (Fork (4) (5))))))
```

(f)
```
NonEmpty(One(Zero(Base(Fork (Node 1 (2) (3))
                            (Node 4 (5) (6))))))
```

Con los ejemplos y casos presentados en este capítulo, hemos mostrado la utilidad del proceso de manufactura de tipos. De está manera se ha cumplido el propósito principal de este trabajo, el cual damos por terminado.

Conclusiones

En este trabajo se describió a detalle un proceso de diseño de estructuras de datos utilizando la programación funcional pura, partiendo de especificaciones que contienen restricciones de tamaño o de forma para los elementos de dichas estructuras. El trabajo se basó principalmente en el artículo *Manufacturing datatypes* de Ralf Hinze [3]. Nuestra aportación consistió en la definición, explicación y demostración de algunos conceptos y propiedades de importancia utilizados en dicho artículo pero no desarrollados a profundidad en el mismo. Por ejemplo, dimos una definición formal de multiconjunto y la utilizamos para demostrar propiedades necesarias; explicamos cómo el concepto intuitivo de tipo contenedor se captura mediante la noción matemática de funtor y demostramos la correspondencia entre multiconjuntos y funtores, que sostiene el proceso de manufactura de tipos. Algunos detalles de la demostración presentada aquí, proporcionan mayor claridad a la misma y están ausentes en el tratamiento original. Finalmente, queremos enfatizar nuestra definición pormenorizada del proceso de manufactura desde la especificación de una estructura de datos hasta la definición de un tipo de datos correspondiente en Haskell.

Para tener una idea clara de los resultados de este proceso, fue de gran importancia estudiar casos particulares no triviales, como los árboles de Braun y los árboles híbridos. Basándonos en una simple observación del artículo original, nos servimos ampliamente del sistema númerico binario para facilitar el entendimiento del proceso de definición y construcción de esta clase de árboles.

Queremos terminar nuestra exposición, mencionando algunas líneas de trabajo futuro relacionadas a lo discutido en este trabajo:

- La definición de bibliotecas de funciones que permitan el uso de las estructuras de datos aquí definidas. Está tarea no es trivial en los

casos donde el tipo obtenido en HASKELL es un tipo anidado, como en los árboles híbridos o de Braun, y requieren del uso de conceptos avanzados de programación funcional, como la recursión polimórfica.

- Manufactura de otras estructuras de datos, tales como: listas de acceso aleatorio, montículos, y distintas clases de árboles como: 2-3 árboles y R-R árboles.
- La verificación formal o certificación del proceso de manufactura de tipos de datos utilizando herramientas de vanguardia, como el asistente de prueba COQ.

Apéndice A

Implementación en Haskell

A.1. Listas finitas

```
data List a = Nil | Cons a (List a) deriving Show

main = putStrLn "Cons 1(Cons 2 (Cons 3 (Nil)))"
```

A.2. Números naturales

```
data Nat a = Zero | Succ (Nat a) deriving Show

main = putStrLn "Succ(Succ(Succ(Succ(Zero))))"
```

A.3. Números pares

```
data Even a = NilE | ConsE (Even a) a a deriving Show

main = putStrLn "ConsE (ConsE (NilE) 3 4) 1 2"
```

A.4. Potencias de tres

```
data P3 a = U a | T (P3 (a, a ,a)) deriving Show

main = putStrLn "T(U((1,2,3),(4,5,6),(7,8,9)))"
```

A.5. Matrices de Toeplitz

```
data Toeplitz a = Corner a
                | Extend (Toeplitz a) a a deriving Show

main = putStrLn "Extend(Extend(Extend(Corner 1) 2 3) 4 5)"
```

A.6. Matrices cuadradas

```
type SquareM a  = SquareMfrom Nil a
data Nil a = Nil
data Cons m a = Cons a (m a)
data SquareMfrom m a = Zero (m (m a))
                     | Succ (SquareMfrom (Cons m) a)

main = putStrLn "Succ(Succ(Zero(Cons(Cons 1(Cons 2 NilM))
                   (Cons(Cons 3(Cons 4 NilM))(NilM)))))"
```

A.7. Árboles de Braun

A.7.1. Implementación propuesta

```
data Empty a = E
data Bin' t a = Bin' a (t a)
data Bin t1 t2 a = Bin a (t1 a) (t2 a)

type Braun = BraunFrom Empty
data BraunFrom t a = Base(t a)
                   | One(BraunFrom (Bin t t ) a)
                   | Two (BraunFrom (Bin (Bin' t) t) a)

main = putStrLn "Two(Base(Bin 1 (Bin' 2 Unit) Unit))"
```

A.7.2. Implementación de Hinze

```
data Empty a = E
data Bin' t a = Bin' a (t a)
data Bin t1 t2 a = Bin a (t1 a) (t2 a)

type Braun = Braun' (Empty) FId
```

```
data Braun' t t' a = Base'(t a)
                   | One' (Braun' (Bin t t ) (Bin t' t) a)
                   | Two' (Braun' (Bin t' t) (Bin t' t') a)

main = putStrLn "Two(Base(Bin 1 2 Unit))"
```

A.8. Árboles híbridos

```
data Fork t a = Fork (t a) (t a)
data Node t a = Node a (t a) (t a)
data Vector a = Empty | NonEmpty (Vector' FId a)
data Vector' t a = Base (t a) | Zero (Vector' (Fork t) a)
                              | One (Vector' (Node t) a)

main = putStrLn "NonEmpty(One(Base(Node 1 (2) (3))))"
```

Bibliografía

[1] Backhouse, R., Jansson, P., Jeuring, J. y Meertens, L. *Generic Programming: An Introduction*En S.D. Swierstra et al. (Eds.): Advanced Functional Programming, Lecture notes in computer science 1608, pp. 28–115, Springer 1999.

[2] Davey, B. A. y Priestley, H. A., *Introduction to Lattices and Order*, Cambridge University Press, segunda edición, 2002.

[3] Hinze, Ralf, *Manufacturing datatypes.* J. Funcional Programming II, pp. 493-524, Septiembre, 2001

[4] Hinze, R., Hackett, J. y James, Daniel W.H, *Functional Pearl: F for Functor.* 2012

[5] Hutton, Graham, *Programming in Haskell*, University of Nottingham, 2007

[6] Mehta, Dinesh P. y Sahni Sartaj, *Handbook of Data Structures and Applications*, 2005

[7] Singh, D., Ibrahim, A. M., Yohanna ,T. y Singh, J. N. *An overview of the applications of multisets*, Novi Sad J. Math, Vol 37, Nº 2, pp. 73-92, 2007.