Universidad Nacional Autónoma de México

Facultad de Ciencias

Listas de acceso aleatorio

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

FRANCISCO JAVIER ENRÍQUEZ LÁVIDA

DIRECTOR DE TESIS:
DR. FAVIO EZEQUIEL MIRANDA PEREA

2015

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

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

1. Preliminares 1
1.1. Listas 2
1.2. Arreglos 3
1.3. Árboles 4
1.4. Acerca del acceso aleatorio 6

2. Sistemas numéricos 9
2.1. Sistemas numéricos no posicionales 9
2.2. Sistemas numéricos posicionales 10
2.3. Representación densa y dispersa 11
2.4. Números binarios sesgados 14
2.5. Representaciones numéricas 18

3. Listas de acceso aleatorio 21
3.1. Implementación binaria estándar 21
3.2. Implementación sin ceros 28
3.3. Implementación binaria sesgada 29

4. Tipos de datos anidados 33
4.1. LAA con tipos de datos anidados 36
4.2. LAA de orden superior 39

Conclusiones 47

Prefacio

Este trabajo está basado en los artículo *Numerical representations as Higher Order Nested Datatypes* de Ralph Hinze y *Purely functional random-access lists* de Chris Okasaki.

En programación funcional las listas son la estructura de datos más común. Su definición es simple y resultan ser una herramienta eficiente en muchos aspectos. Sin embargo, su principal desventaja consiste en que las operaciones de acceso son de orden lineal, es decir, accesar al i-ésimo elemento requiere $O(i)$ tiempo. Por esta razón, en algunas situaciones, el programador funcional añora la eficiencia de los arreglos imperativos. Los arreglos son una estructura inherentemente imperativa, que resulta difícil de implementar en un marco funcional debido a que en este ámbito, las estructuras de datos son persistentes, es decir, la versión previa de un arreglo debe seguir disponible aun después de una actualización.

En este trabajo, presentamos diversas implementaciones de una estructura de datos funcional llamada *Lista de acceso aleatorio*, la cual permite definir las operaciones de búsqueda y actualización en tiempo logarítmico. Más aún, en una de nuestras implementaciones, las funciones primitivas de lista conservan su eficiencia ejecutándose en tiempo constante, por lo que en cualquier aplicación podemos sustituir el uso de listas comunes mediante listas de acceso aleatorio, conservando o mejorando la eficiencia.

El diseño e implementación de nuestras listas de acceso aleatorio se sirve de la idea de representación numérica, la cual consiste en definir una estructura de datos siguiendo una fuerte analogía con un sistema numérico, a saber, la representación de n en un sistema numérico genera una estructura de tamaño n.

El contenido del trabajo es como sigue: En el capítulo 1 se desarrollan los preliminares sobre estructuras de datos funcionales, en particular, listas y árboles. Los sistemas y representaciones numéricas se discuten en el capítulo 2, en particular nos interesan distintos sistemas binarios, los cuales dan lugar a distintas implementaciones de las listas de acceso aleatorio, desarrolladas en el capítulo 3. El problema de capturar estáticamente una lista de acceso aleatorio válida, se resuelve en el capítulo 4 mediante conceptos avanzados de la programación funcional, en particular, mediante el uso de tipos de datos anidados. Finalmente ofrecemos algunas conclusiones y apuntamos ciertos temas como posibles líneas de trabajo futuro.

Capítulo 1

Preliminares

Actualmente en las ciencias de la computación existen diversos paradigmas de lenguajes de programación, siendo hoy en día el más popular el paradigma orientado a objetos, descendiente directo del paradigma imperativo. No obstante, el paradigma funcional resulta ser una herramienta muy poderosa en la programación debido a que una función en este ámbito es realmente una función en el sentido matemático. Es decir, una funcion f declarada en un lenguaje de programación funcional puro, siempre arrojará, ante una entrada x, el mismo resultado. Esto es, no existen *efectos colaterales* que puedan alterar el comportamiento de la función. Esta propiedad hace mucho más fácil entender el comportamiento de un programa y razonar acerca de sus consecuencias.

Algo fundamental en la programación funcional es que las estructuras de datos definidas son automáticamente persistentes. ¿Qué significa *persistencia*? Usualmente, cuando se modifica una estructura de datos en el paradigma imperativo, nos es indiferente que la versión anterior de la estructura sea inaccesible después de la modificación. Sin embargo, en la programación funcional, se espera que ambas versiones estén disponibles para procesamiento posterior. Una estructura de datos que permite tener varias versiones de sí misma es llamada *persistente*, mientras aquellas que sólo permiten una única versión son llamadas *efímeras*. Las estructuras definidas en un lenguaje de programación imperativo son, por lo regular, efímeras así que cuando se requiere una estructura de datos persistente, dichas estructuras son significativamente más complicadas que las versiones efímeras.

Es importante destacar también que, a pesar de que hablaremos sobre los tiempos de ejecución de las funciones sobre las distintas estructuras que daremos a conocer, en el análisis hecho no profundizaremos demasiado, puesto que un análisis detallado de la complejidad va más allá de nuestros objetivos.

A lo largo de este documento, utilizaremos el lenguaje de programación Haskell. Se comentarán ciertos aspectos generales de este lenguaje. Sin embargo, se asumen conocimientos básicos sobre el mismo. Para una introducción más detallada de este lenguaje, veáse por ejemplo [3]. Además, destacamos que para la interpretación del código, utilizamos el intérprete Hugs.

Empezaremos discutiendo brevemente la estructura de datos "*lista*", la cual es una herramienta fudamental en la programación funcional.

1.1. Listas

Una lista es una estructura que almacena datos en forma similar a los conjuntos, pero tomando en cuenta el orden y el número de presencias de cada elemento. Por ejemplo: la lista [1,2,3] es distinta a la lista [2,3,1] y la lista [4,4,5] es distinta a la lista [4,5].

Nos interesan especialmente tres operaciones `cons`, `head` y `tail`, cuyo funcionamiento se explica a continuación:

- `cons`: Añade un elemento al principio de una lista dada.
- `head`: Obtiene la *cabeza* de una lista, es decir, el primer elemento de la misma.
- `tail`: Obtiene la *cola* de una lista, es decir, la lista resultante al quitar el primer elemento de la lista de entrada.

La implementación de las listas en Haskell la haremos utilizando un constructor para una lista vacía y uno más para unir un elemento con otra lista. La lista vacía la representaremos con el constructor `Nil` y el constructor para unir lo llamaremos `Cons`. Así nuestra implementación es la siguiente:

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

De esta manera, las operaciones antes discutidas se implementan fácilmente como sigue:

```
cons :: a -> List a -> List a
cons a l = Cons a l

head :: List a -> a
head (Cons a l) = a

tail :: List a -> List a
tail (Cons a l) = l
```

Obsérvese que las funciones `head` y `tail` son parciales, puesto que no están definidas para la lista vacía.

Además de estas operaciones también nos interesará acceder a un elemento aleatorio de la lista, es decir, dado un entero no negativo i, queremos encontrar el elemento en esa posición, considerando que la numeración inicia en 0, i.e., la cabeza siempre corresponde al índice 0. Una vez hallado un elemento, también nos gustaría poder actualizar ese elemento. Estas operaciones las llamaremos *funciones de acceso* definidas como `lookup` y `update`, respectivamente.

La idea de la implementación de `lookup` es la siguiente: Si se busca el elemento 0, devolvemos la cabeza de la lista. En otro caso utilizamos recursión

buscando el elemento $i-1$ sobre la cola de la lista. La idea de `update` es idéntica. No obstante, hay que reconstruir la lista que se ha recorrido anteriormente. Esto lo haremos con llamadas a la función `cons`. El código correspondiente se ve a continuación:

```
lookup :: Int -> List a -> a
lookup 0 l = head l
lookup i l = lookup (i-1) (tail l)

update :: Int -> a -> List a -> List a
update 0 x l = cons x (tail l)
update i x l = cons (head l) (update (i-1) x (tail l))
```

Nótese que las operaciones `cons`, `head` y `tail` tienen un tiempo de ejecución $O(1)$. Sin embargo, las operaciones `lookup` y `update` se ejecutan en $O(n)$. A lo largo de este documento resolveremos este inconveniente de estas listas.

1.2. Arreglos

Si el lector tiene nociones de la programación imperativa, seguramente podrá recordar el concepto de arreglos en dicho paradigma. Los arreglos son una estructura similar a las listas en el sentido de que ambas son una secuencia indexada. No obstante, existe una diferencia fundamental. A diferencia de las listas, una vez definido un arreglo su tamaño es invariante. Es decir, siempre guardará la misma cantidad de elementos. Al tener un tamaño fijo, la programación imperativa aprovecha el concepto de memoria contigua para garantizar que las funciones de acceso se ejecuten en $O(1)$. ¿Cómo funciona esto? Cuando el arreglo es declarado, cierta cantidad de memoria se reserva para guardar los elementos del mismo. Por ejemplo, si sabemos que cada elemento de la lista utiliza cuatro espacios de memoria (*bytes*) y guardaremos exactamente n elementos, al ser declarado el arreglo, se reservan $4n$ posiciones de memoria. Así, cuando queremos acceder al elemento i, basta movernos $4i$ posiciones de memoria a partir de la primera posición del arreglo. Suponiendo que la implementación de la memoria nos permite acceder a sus posiciones en $O(1)$, se puede acceder a cualquier elemento del arreglo en $O(1)$.

Para el caso de los arreglos, nos interesan 3 operaciones básicas: `create`, `lookup` y `update`. Estas operaciones especifican como sigue:

- `create`: Crea un arreglo de tamaño n.

- `lookup`: Obteniene el elemento i de un arreglo.

- `update`: Actualiza el elemento i de un arreglo.

La implementación en el paradigma funcional de un arreglo, en principio, es un tanto complicada dado que sin el concepto de memoria no es inmediato el acceso aleatorio eficiente a los elementos de una estructura. A lo largo de este

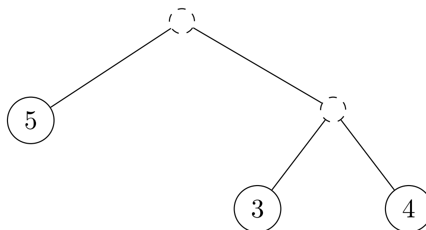
trabajo resolveremos este problema, lo cual dará como resultado una estructura funcional capaz de implementar arreglos.

1.3. Árboles

Para llegar a nuestro objetivo final utilizaremos, entre otras estructuras, árboles binarios. Un árbol binario es una hoja o un nodo con exclusivamente dos hijos, llamados *subárbol izquierdo* y *subárbol derecho*. Así, el tipo de árboles que guardan datos únicamente en sus hojas, en este caso, datos de tipo `a`, tiene la siguiente definición:

```
data Tree a =
    Leaf a
  | Node (Tree a) (Tree a)
```

Los árboles pueden ser utilizados para diversos propósitos y esta es su representación general. Un árbol puede tener un número arbitrario de hojas, cada una con un elemento de tipo `a`. Por ejemplo el árbol de tres elementos de tipo `int` representado por:



Se implementa como

```
(Node (Leaf 5) (Node (Leaf 3) (Leaf 4)) )
```

Notemos también que la disposición de los elementos a lo largo del árbol es totalmente arbitraria.

Definiremos sobre árboles el concepto de *altura*. La altura de un árbol está definida por la distancia más larga entre la *raíz* hasta una hoja. Así, la definición en Haskell está dada por:

```
height :: Tree a -> Int
height (Leaf _) = 0
height (Node t1 t2) = 1 + ( max (height t1) (height t2) )
```

Nos interesará también el número de hojas y nodos internos que tiene un árbol. Las definiciones de ambas operaciones son directas:

```
nleafs :: Tree a -> Int
nleafs (Leaf _) = 1
nleafs (Node t1 t2) = (nleafs t1) + (nleafs t2)
```

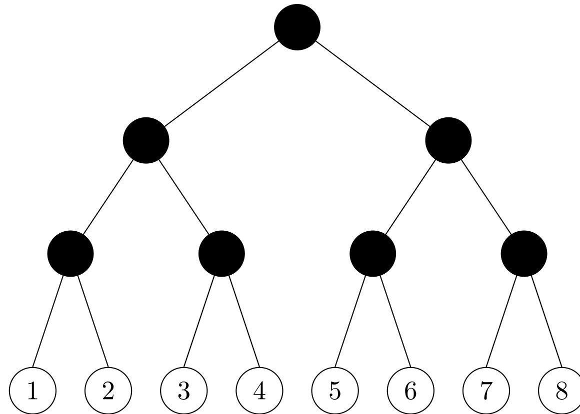
```
nnodes :: Tree a -> Int
nnodes (Leaf _) = 0
nnodes (Node t1 t2) = 1 + (nnodes t1) + (nnodes t2)
```

De estas definiciones se observa que para saber exactamente cuantos nodos u hojas tenemos, hay que recorrer todo el árbol. Así que si estamos guardando información en las hojas únicamente, es decir, los nodos internos no proveen información útil, entonces saber la cantidad de elementos dentro de un árbol puede llegar a ser más ineficiente que recorrer una lista común. Es por ello que añadiremos ciertas restricciones a los árboles.

Puesto que será muy relevante el conocer el número de hojas en un árbol, nos interesa una manera eficiente de calcularlo, para lo cual, utilizaremos árboles binarios perfectos.

Definición 1.3.1 (Árboles binarios perfectos)**.** Un árbol binario perfecto de rango 0 es una hoja. Un árbol binario perfecto de rango $r+1$ se define como un nodo con dos subárboles binarios perfectos de rango r.

Gráficamente, un árbol binario perfecto de rango 3 con datos en las hojas se ve de la siguiente manera:



Ahora, con esta definición, podemos saber exactamente cuantas hojas tiene el árbol sin necesidad de recorrerlo. Bastará con hacer un cálculo según su rango. Para ello damos la siguiente proposición:

Proposición 1.3.1. *Un árbol binario perfecto de rango r tiene 2^r hojas.*

Demostración. Inducción sobre r.
Caso base $r = 0$.
El árbol es una hoja. Es decir el número de hojas del árbol es $1 = 2^0 = 2^r$
Hipótesis de Inducción:
Un árbol de rango r tiene 2^r hojas.
Paso inductivo. Demostrar que un árbol de rango $r+1$ tiene 2^{r+1} hojas.
Sabemos que un árbol de rango r tiene 2^r hojas y un árbol de rango $r+1$ tiene dos subárboles de rango r. Las hojas sólo están en los subárboles, entonces, el árbol de rango $r+1$ tiene $2^r + 2^r$ hojas. Y también $2^r + 2^r = 2(2^r) = 2^{r+1}$ □

Más aún, también podemos saber exactamente cuantos nodos tiene un árbol a través de su rango:

Proposición 1.3.2. *Un árbol binario perfecto de rango r tiene $2^{r+1}-1$ nodos.*

Demostración. Inducción sobre r.
Caso base $r = 0$.
El árbol es una hoja. Una hoja en particular es un árbol con un único nodo. Es decir, el árbol tiene $1 = 2^1 - 1 = 2^{r+1} - 1$ nodos.
Hipótesis de inducción:
Un árbol de rango r tiene $2^{r+1} - 1$ nodos.
Paso inductivo. Demostrar que un árbol de rango $r + 1$ tiene $2^{r+2} - 1$ nodos. Sabemos que un árbol de rango r tiene $2^{r+1} - 1$ nodos. Así que al juntar dos árboles tenemos $2(2^{r+1} - 1) + 1$. Añadimos un nodo ya que al unirlos creamos un nuevo nodo. Vemos que:

$$\begin{aligned} 2(2^{r+1} - 1) + 1 &= 2^{r+2} - 2 + 1 \\ &= 2^{r+2} - 1 \end{aligned}$$

□

Al cumplir estas propiedades de tamaño, los árboles binarios perfectos, en conjunto con las listas comunes serán de vital importancia para agilizar el acceso a elementos arbitrarios en una lista. En la siguiente sección discutiremos como es que los árboles nos permiten el fácil acceso a los elementos de una estructura.

Antes de continuar, remarcaremos el hecho de que en todo momento nos será de suma importancia saber el número de nodos y/o hojas que contiene un árbol. Éste número es calculado de manera inmediata conociendo el tamaño del árbol. No obstante, con nuestra definición actual no es posible conocer el rango del árbol sin tener que recorrerlo. Es por ello que a nuestra estructura le añadiremos un entero que indicará el rango del árbol. Con esta modificación, la implementación será:

```
data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)
```

Con esta implementación, definiremos la función `size` que nos permitirá conocer el número de hojas que tiene el árbol. Esta función simplemente devolverá 2^i, con i el rango del árbol. Esta es su implementación:

```
size :: Tree a -> Int
size (Leaf _) = 1
size (Node n _ _) = 2^n
```

Dicho esto, podemos discutir como es que los árboles binarios perfectos facilitan el acceso a elementos de una estructura.

1.4. Acerca del acceso aleatorio

El problema de añadir y quitar elementos al inicio de una lista lo tenemos prácticamente resuelto con la definición de listas comunes. Sin embargo, no es

inmediato adaptar estas operaciones para agilizar el acceso a elementos aleatorios en dicha estructura. Es por ello que primero nos enfocaremos en resolver el problema del acceso aleatorio.

Si suponemos momentáneamente que nuestra lista va a tener exactamente 2^i elementos para algún valor dado de i, entonces, utilizando un árbol binario perfecto para implementar la lista, se satisface nuestra necesidad del rápido acceso a los elementos con información útil de la estructura, ya que podemos lograr un acceso logarítmico como se explica a continuación. Primero, convenimos en guardar los elementos de izquierda a derecha en el árbol, es decir, la hoja más a la izquierda del árbol contendrá el primer elemento de la lista y la hoja más a la derecha, al último. Como segundo hecho, démonos cuenta que, con los elementos guardados de esta manera, dado un árbol de tamaño 2^i, el subárbol izquierdo contiene los primeros 2^{i-1} elementos y su subárbol derecho contiene los 2^{i-1} elementos restantes.

A partir de estos hechos, podemos implementar las funciones de acceso y actualización de la siguiente manera: Si el elemento buscado i es menor o igual que 2^{k-1}, con k el rango del árbol, aplicamos recursión con el subárbol izquierdo buscando el mismo elemento i. En otro caso, buscamos el elemento $i - 2^{k-1}$ en el subárbol derecho. Restamos 2^{k-1} porque esto es el número de elementos que recorrimos al descartar el subárbol izquierdo. Esto es una búsqueda binaria común que sabemos se ejecuta en $O(\log n)$.

Siguiendo esta idea, la implementación de las operaciones de acceso y actualización en HASKELL es la siguiente:

```
lookupTree :: Tree a -> Int -> a
lookupTree (Leaf x) 0 = x
lookupTree t@(Node _ t1 t2) i
    | i < (size t) `div` 2 = lookupTree t1 i
    | otherwise = lookupTree t2 (i - ((size t) `div` 2))


updateTree :: Tree a -> Int -> a -> Tree a
updateTree (Leaf x) 0 y = Leaf y
updateTree t@(Node n t1 t2) i y
    | i < (size t) `div` 2 = Node n (updateTree t1 i y) t2
    | otherwise = Node n t1 (updateTree t2 (i - ((size t) `div` 2))
        y)
```

Ahora bien, esta implementación solo sirve cuando lo que deseamos es implementar una lista de tamaño 2^k. Esto no es conveniente para el caso general dado que deseamos que nuestras listas tengan tamaño arbitrario. ¿Cómo reconciliar el tamaño arbitrario con el acceso aleatorio?

La primera alternativa para implementar una lista de tamaño arbitrario n, es escoger un árbol de rango k mínimo, tal que $n \leq 2^k$. Esta opción no nos es conveniente dado que si nuestra estructura tuviese $n = 2^k + 1$ elementos, tendríamos que usar un árbol de tamaño 2^{k+1}, por lo que se estarían desperdiciando $2^k - 1$ espacios de almacenamiento. La alternativa que utilizaremos será el guardar una colección de árboles cuyos tamaños sumados sean exactamente n. Esta colección existe trivialmente si utilizamos n árboles de rango 0. Sin embargo, esto sería equivalente a la implementación usual de listas. No obstante,

recordemos que cualquier número tiene una expansión binaria de acuerdo a la siguiente proposición:

Proposición 1.4.1. $\forall n \geq 1 \in \mathbb{N},\ \exists\ S \subset \{2^i \mid i \in \mathbb{N}\}$, *tal que* $\sum_{x \in S} x = n$.

Demostración. Inducción sobre n
Caso base $n = 1$**.**
Sea $S = \{2^0\} = \{1\}$. Claramente $\sum_{x \in S} x = 1$
Hipótesis de inducción: $S \subset \{2^i \mid i \in \mathbb{N}\}$ y $\sum_{x \in S} x = n$
Paso inductivo. Demostrar que existe $R \subset \{2^i \mid i \in \mathbb{N}\}$ tal que

$$\sum_{x \in R} x = n + 1$$

Sean $k = \text{mín}\{i \,|\, i \in \mathbb{N},\, 2^i \notin S\}$ y $R = S \backslash \{2^i \,|\, i < k\} \cup \{2^k\}$.
Obsérvese que, por la definición de k,

$$\{2^i \,|\, i < k,\, 2^i \in S\} = \{2^i \,|\, i < k,\, i \in \mathbb{N}\}$$

Ahora

$$\begin{aligned}
\sum_{x \in R} x &= (n - \sum_{i=0}^{k-1} 2^i) + 2^k \\
&= (n - \sum_{i=0}^{k-1} 2^i + (1 - 1)) + 2^k \\
&= n - ((\sum_{i=0}^{k-1} 2^i) + 1) + 2^k + 1 \\
&= n - 2^k + 2^k + 1 && \text{Puesto que } \sum_{i=0}^{k-1} 2^i = 2^k - 1. \\
&= n + 1
\end{aligned}$$

□

La proposición anterior garantiza que cualquier número n tiene una expansión en potencias de 2, lo cual se traduce a que cualquier lista de tamaño n puede implementarse mediante una colección de árboles binarios perfectos. Falta decidir como guardaremos esta colección de árboles y en qué orden serán almacenados. También debemos dar un proceso para añadir y quitar elementos de un árbol. Profundizaremos en ello en el siguiente capítulo.

Capítulo 2

Sistemas numéricos

Cuando utilizamos números naturales usualmente los representamos con combinaciones de los dígitos del 0 al 9. Esto corresponde a la representación en el sistema numérico decimal. Este sistema no es el único que existe. En este capítulo utilizaremos diversos sistemas para representar los naturales, los cuales nos llevarán a diversas implementaciones de estructuras de datos, conocidas como representaciones numéricas.

2.1. Sistemas numéricos no posicionales

Cuando éramos infantes nos enseñaron a contar con los dedos o utilizando "palitos". Entonces para representar el número n teníamos n palitos, no importaba el orden de los palitos ya que todos los palitos eran iguales. Este concepto está formalizado utilizando la siguiente definición:

- El cero es un número natural.
- El sucesor de un número natural es un natural.

Con esta definición, la analogía es: El cero representa no tener palitos y aumentar un palito es equivalente a obtener el sucesor de la cantidad de palitos que teníamos antes. La definición formal en HASKELL es:

```
data Nat =  Zero
          | Succ Nat
```

Ahora bien, para nuestros propósitos requerimos las operaciones de suma y sucesor para estos números. La operación sucesor es inmediata a partir de la definición del tipo ya que queda implementada mediante el constructor `Succ`. Para la función suma nos servimos de la siguiente definición recursiva. Sumarle cero a un número resulta en el mismo número, y sumarle el sucesor de m a n es equivalente a obtener el sucesor de la suma de n y m. Es decir: $n + (\texttt{Succ}\ m) = \texttt{Succ}(n + m)$. Definiéndolo formalmente en HASKELL obtenemos:

```
add :: Nat -> Nat -> Nat
add n Zero = n
add n (Succ m) = Succ (add n m)
```

Ahora bien, puesto que la construcción del sucesor de un número se realiza de manera sencilla añadiendo un constructor `Succ`, la definición del predecesor se hace de manera directa eliminándolo:

```
pred :: Nat -> Nat
pred (Succ n) = n
pred Zero = error "Predecesor of zero"
```

Nótese que el predecesor no está definido en cero. Podríamos manejar el error definiendo como `Zero` al predecesor de él mismo. Sin embargo, para propósitos de este documento, dejaremos este caso como un error.

A simple vista un sistema numérico no posicional es muy rudimentario, pero más adelante veremos como una estructura de datos muy común corresponde a una representación de este sistema.

2.2. Sistemas numéricos posicionales

Retomando el sistema numérico decimal tradicional es claro ver que, por ejemplo, el número 123 no es igual al número 231 ni al número 321 aunque los tres utilicen los mismos dígitos y en la misma cantidad. Esto se debe a que el sistema numérico decimal es un sistema numérico posicional, es decir, el dígito tiene distinto peso según su posición en la sucesión de dígitos.

En general, un sistema numérico posicional es un par $\mathcal{S} = \langle D, W \rangle$ donde:

- $D \neq \varnothing$ es un conjunto cuyos elementos llamaremos dígitos. Usualmente D es un subconjunto de los naturales del 0 al 9.
- $W \neq \varnothing$, $W \subset \mathbb{R}$ es un conjunto de pesos, usualmente infinito.

Una vez fijado un sistema numérico $\mathcal{S} = \langle D, W \rangle$, una sucesión de dígitos $d_0 d_1 d_2 \ldots d_n$, donde $d_i \in D$, representa al número dado por:

$$\sum_{i=0}^{n} d_i \cdot w_i$$

Introducimos la notación $n \approx d_0 d_1 d_2 \ldots d_k$ que significará que el entero n está representado por $d_0 d_1 d_2 \ldots d_k$, es decir, $n = \sum_{i=0}^{k} d_i \cdot w_i$

Es importante recalcar que, a lo largo de este documento, consideraremos que si $n \approx d_0 d_1 d_2 \ldots d_k$, entonces d_0 es el dígito menos significativo y d_k es el dígito más significativo de n, es decir, d_0 tendrá asociado el peso más bajo y d_k el más alto. Esta convención es a la inversa de los sistemas numéricos tradicionales. La cual es común en el estudio de los sistemas numéricos en abstracto y seguimos aquí por simplicidad.

Veamos algunos ejemplos de sistemas numéricos:

- Sistema numérico decimal común: $D = \{0, \ldots, 9\},\ W = \{10^i \,|\, i \in \mathbb{N}\}$.
- Sistema binario común: $D = \{0, 1\},\ W = \{2^i \mid i \in \mathbb{N}\}$. De modo que $5 \approx 101$.
- Sistema binario con $1, 2$ (sin cero): $D = \{1, 2\},\ W = \{2^i \mid i \in \mathbb{N}\}$. De modo que $5 \approx 12$.
- Sistema binario con $0, 1, 2 :$ $D = \{0, 1, 2\},\ W = \{2^i \mid i \in \mathbb{N}\}$. De modo que $5 \approx 12$, $5 \approx 101$.
- Sistema binario de Fibonacci: $D = \{0, 1\},\ W = \{F_i \mid i \in \mathbb{N}\}$ donde F_i es el i-ésimo número de Fibonacci $(1, 1, 2, 3, 5, 8, 13, \ldots)$. De modo que $5 \approx 00001$, $5 \approx 0011$, $7 \approx 00101$.

Notemos que en algunos sistemas la representación de un número no es única. Cuando esto suceda, diremos que el sistema es *redundante*

2.3. Representación densa y dispersa

Con los sistemas posicionales, hay dos maneras distintas de representar un número natural: la representación *densa* y la representación *dispersa*. En la primera haremos explícitos los dígitos nulos. En cambio, en la segunda omitiremos estos dígitos. Para ello, la representación deberá tener algún indicador de los pesos de cada dígito distinto de cero.

Para el sistema binario común, la representación usual es una representación *densa*, por ejemplo $14 \approx 0111$. Por otro lado, una representación dispersa se obtiene al representar un número mediante la lista de los pesos de los dígitos no nulos. Por ejemplo el $14 \approx [2,\ 4,\ 8]$.

Para nuestro propósito final nos serán de especial interés ambas formas de representación para sistemas binarios. Empezaremos analizando más a fondo los primeros.

Empecemos definiendo una representación densa en Haskell:

```
data Digit = One
           | Zero

type Nat = [Digit]
```

De esta manera, un número es representado por una lista de `Zero` y `One`. Por ejemplo el 14, quedaría implementado mediante la lista `[Zero, One, One, One]`. En seguida discutiremos la implementación de las operaciones básicas (incremento y decremento). Para obtener el incremento en uno, o sucesor de un número basta fijarnos en los casos posibles: Si tenemos una sucesión vacía de dígitos, la cual representa al 0, deberemos regresar una sucesión incluyendo únicamente al dígito `One`; si tenemos una sucesión que inicia con `Zero`, solo sustituimos éste por el dígito `One`. El caso interesante se da cuando la lista inicia con `One`. Recordemos cuando aprendimos a sumar, dejando de un lado que el orden de sumar era de derecha a izquierda, podremos recordar que cuando sumábamos 1

a un 9, colocábamos un 0 y *llevábamos* 1, este 1 debía sumarse al resultado de la suma de los siguientes dígitos. Consideremos la siguiente figura:

```
  19
+ 11
----
  30
```

Empezamos sumando 9 y 1. Colocamos un 0 pero llevamos 1. Luego procedemos a sumar el par de unos así que deberíamos colocar un 2 pero llevábamos 1, por lo tanto colocamos el 3. Esta operacion de *llevar 1* es conocida como operación de *acarreo*.

Esta es la misma operación que debemos ejecutar cuando queremos añadir 1 a una sucesión de dígitos que inicia con `One`. Solo que lo estamos realizando en orden inverso (de izquierda a derecha) dado que el dígito menos significativo está a la izquierda. Por lo tanto, cuando tenemos una sucesión que inicia con `One` colocamos un `Zero` y añadimos `One` al resto de la sucesión. La implementación queda así:

```
inc :: Nat -> Nat
inc [] = [One]
inc (Zero : ds) = One : ds
inc (One : ds) = Zero : (inc ds)
```

Para decrementar la idea es similar, los casos de sucesiones que inician con `One` son sencillos. Basta sustituir el `One` con `Zero` o regresar la lista vacía cuando la sucesión sólo tenía un `One`. Cuando la sucesión empieza con `Zero` hay que realizar una operación similar al acarreo. Recordemos nuestra infancia con el siguiente ejemplo:

```
  20
- 11
----
  09
```

Al restarle 1 al 0, no tenemos alternativa más que poner el dígito 9 y decir: *Le pedimos prestado* al siguiente dígito, que en este caso particular es 2. Así cuando restamos 1 al 2, el resultado no es 1, porque le habíamos pedido prestado 1 al 2, es decir, el 2 se convierte en 1 por lo que en realidad restamos 1 a 1 y el resultado es 0. Así para nuestra implementación, cuando restemos 1 a una sucesión que inicia con `Zero`, sustituimos el `Zero` por `One` y le pedimos prestado al resto de la sucesión, i.e., la decrementamos en 1. La implementación es:

```
dec :: Nat -> Nat
dec [] = error "Decrement of nil"
dec [One] = []
dec (One : ds) = Zero : ds
dec (Zero : ds) = One : (dec ds)
```

Pasemos ahora al caso de la implementación dispersa. Representamos un número binario mediante su lista ascendente de pesos. Definiremos las mismas operaciones para estos sistemas numéricos.

Añadir 1 a una sucesión de pesos es fácil si ésta no contiene el peso 1, pero si lo contiene ¿qué hacemos? La solución consiste en darnos cuenta que tenemos dos pesos 1, los cuales pueden sustituirse por un solo peso 2. Entonces, en lugar de añadir el peso 1 a la lista que ya contiene un peso 1, quitamos dicho peso de la sucesión y agregamos el peso 2. Si la sucesión ya contenía un peso 2, procedemos igual: quitamos el peso 2 y añadimos un peso 4, y así sucesivamente hasta que no sea necesario juntar los pesos.

Decrementar una sucesión es similar. Si la sucesión contiene el peso 1, lo quitamos y habremos terminado. Si no es así, debemos obtener un peso 1 del resto de la sucesión. ¿Cómo obtenemos un peso 1? La manera de obtener un 1, es obtener un peso 2 de la sucesión y descomponerlo en dos pesos 1. Dejamos un peso 1 en la sucesión y el otro lo quitamos; así habremos restado 1 a la sucesión. ¿Qué ocure cuando la sucesión no tiene un peso 2 para ser descompuesto en 2 pesos 1? Entonces deberemos obtener ese peso 2 de la sucesión descomponiendo un peso 4 en dos pesos 2; si no hay un peso 4, buscamos un peso 8 para descomponerlo en dos pesos 4; y así sucesivamente.

La implementación dispersa completa en HASKELL es:

```
type Nat = [Int]

carry :: Int -> Nat -> Nat
carry w [] = [w]
carry w ws@(w1 : ws1) = if w < w1 then w : ws
                        else carry (w*2) ws1

borrow :: Int -> Nat -> Nat
borrow w ws@(w1 : ws1) = if w == w1 then ws1
                         else w : (borrow (2*w) ws1)

inc :: Nat -> Nat
inc ws = carry 1 ws

dec :: Nat -> Nat
dec ws = borrow 1 ws

check :: Nat -> Bool
check [] = True
check l = checkAux l (takeWhile (<= m) powersOfTwo)
      where m = maximum l
```

Es inmediato ver que nuestra implementación tiene un problema, puesto que un número se representa mediante una lista arbitraria de enteros, los cuales no son necesariamente potencias de 2. Para intentar solucionar este problema, definimos la función `check` que decidirá si una lista de enteros es un natural válido o no. Para ello, tenemos dos posibles casos:

- Una lista vacía es un natural válido dado que ésta representa al 0.

- Una lista no vacía es válida si sus elementos son potencias de 2 en orden ascendente sin repeticiones.

Para implementar esto, hacemos uso de dos funciones auxiliares: `powersOfTwo` y `checkAux`. La primera únicamente devuelve una lista infinita de las potencias de 2, lo cual es posible gracias a la evaluación perezosa. La segunda, recibirá la lista de dígitos sin verificar y una lista de potencias de 2 finita. Al inicio, los dígitos sin verificar serán todos los dígitos, y la lista de potencias de 2 todas las potencias de 2 menores o iguales que m, donde m es el elemento más grande de la lista por verificar. Ahora bien, para una lista no vacía, se verificará que la cabeza sea un elemento de la lista de potencias de 2 finita. Si no es potencia de 2, es claro que el natural no es válido. Si lo es, volvemos a aplicar `checkAux` con los dígitos restantes y la misma lista de potencias de 2, quitando los elementos menores o iguales que el primer dígito.

Así, la definición en Haskell será:

```
powersOfTwo :: [Int]
powersOfTwo = 1 : (map (\x->2*x) powersOfTwo)

checkAux :: Nat -> [Int] -> Bool
checkAux [] _ = True
checkAux _ [] = False
checkAux (x:xs) pows =
    elem x pows && checkAux xs (dropWhile ( <= x) pows)
```

Más adelante mostraremos una implementación que utiliza tipos de datos más sofisticados, lo cual evitará la necesidad de verficar si una lista es válida o no.

2.4. Números binarios sesgados

A continuación presentaremos un sistema numérico binario que nos resultará muy útil para nuestro objetivo final, dado que las operaciones de incremento y decremento son más eficientes. Aunque los conjuntos de pesos y dígitos se vuelven más complejos.

Cambiaremos el conjunto de pesos $\{2^i \mid i \in \mathbb{N}\}$ por el conjunto $W = \{2^{i+1} - 1 \mid i \in \mathbb{N}\}$. También, utilizaremos otros dígitos, a saber $D = \{0, 1, 2\}$. A primera instancia podemos ver que este sistema es redundante. Por ejemplo el número decimal 8 puede ser representado como 220 ó 101.

Para quitar esta ambigüedad añadiremos una restricción más: *El dígito 2 sólo puede aparecer una vez y debe ser el primer dígito distinto de cero*. Cuando un número binario sesgado cumple esta última restricción diremos que está en su forma *canónica*. Empecemos mostrando que la representación canónica existe y es única.

Lema 2.4.1. *Sea α un número binario sesgado en su representación canónica. Si α tiene n dígitos y $\alpha \approx A$ entonces $2^n - 1 \leq A \leq 2^{n+1} - 2$.*

Demostración. Primero veamos que:
$2^n - 1 \leq A$
Es claro que el número más pequeño que se puede representar con n dígitos es:

$$\beta = 0 \ldots 01$$

Por lo que, como $\beta \approx 2^{n-1}$, se sigue que $2^{n-1} \leq A$.
Ahora veamos que se cumple que $A \leq 2^{n+1} - 2$.

De manera opuesta a la desigualdad anterior, buscaremos una sucesión de n dígitos β con valor máximo. En tal caso $\beta = 0^k 2 1^j$, con $j + k + 1 = n$.
Pero observemos también que:
$0^k 2 1^j \approx \sum\limits_{i=k+2}^{n} (2^i - 1) + 2(2^{k+1} - 1)$
$= \sum\limits_{i=k+2}^{n} 2^i - \sum\limits_{i=k+2}^{n} 1 + 2(2^{k+1} - 1)$
$= \sum\limits_{i=0}^{n} 2^i - \sum\limits_{i=0}^{k+1} 2^i - \sum\limits_{i=k+2}^{n} 1 + 2(2^{k+1} - 1)$
Además recordemos que: $\sum\limits_{i=0}^{n} 2^i = 2^{n+1} - 1$
Así: $0^k 2 1^j \approx (2^{n+1} - 1) - (2^{k+2} - 1) - (n - (k + 2)) + 2(2^{k+1} - 1)$
$= (2^{n+1} - 1) - 2^{k+2} + 1 - n + k + 2 + 2^{k+2} - 2$
$= 2^{n+1} - 1 - (n - k)$
$\leq 2^{n+1} - 2$ Dado que $k \leq n - 1$
De esto podemos concluir que $\beta = 0^n 2$ y además $\beta \approx 2^{n+1} - 2$ □

Lema 2.4.2. *Si $A \leq 2^{n+1} - 2$ entonces existe α tal que $\alpha \approx A$.*

Demostración. Inducción sobre n. Base $n = 1$. $A \leq 2^2 - 2 = 2$ entonces $A \in \{0, 1, 2\}$. Si $A = 0$, entonces α es la secuencia vacía y como $1 \approx 1$ y $2 \approx 2$, la afirmación es válida.
Hipótesis de inducción. Si $A \leq 2^{n+1} - 2$, entonces existe α tal que $\alpha \approx A$.
Supongamos ahora que $A \leq 2^{n+2} - 2$. Entonces, tenemos 2 casos:

- $A = 2^{n+2} - 2$. Sea $\alpha = 0^n 2$. Así $\alpha \approx 2(2^{n+1} - 1) = 2^{n+2} - 2$.

- $A \leq 2^{n+2} - 3$.

  Por la hipótesis de inducción, si $A \leq 2^{n+1} - 2$, α existe. Por lo tanto, basta analizar el caso cuando $2^{n+1} - 1 \leq A \leq 2^{n+2} - 3$.

  Sea $B = A - 2^{n+1} + 1$. Es fácil ver que $B \leq 2^{n+1} - 2$. De donde, por la HI, existe β tal que $\beta \approx B$. Además, por el lema anterior, como $B \leq 2^{n+1} - 2$ entonces β tiene k dígitos con $k \leq n$.

  Así proponemos la sucesión: $\alpha = \beta 0^{n-k} 1$. Veamos pues que $\alpha \approx A$.

  $\alpha = \beta 0^{n-k} 1 \approx B + 2^{n+1} - 1 = A - 2^{n+1} + 1 + 2^{n+1} - 1 = A$. $\therefore \alpha = \beta 0^{n-k} 1 \approx A$

□

Proposición 2.4.3. *Sean α y β representaciones canónicas. Si $\alpha \approx A$ y $\beta \approx B$ y $\alpha \neq \beta$, entonces $A \neq B$.*

Demostración. Para efectos de esta demostración, utilizaremos 2 notaciones no mencionadas anteriormente. La notación $|\alpha|$ significará la cantidad de dígitos de α y la notación $[\alpha]$ será el número representado por α, es decir, si $\alpha \approx A$, entonces $[\alpha] = A$.

Caso 1. $|\alpha| \neq |\beta|$

SPG, supongamoes que $|\beta| + 1 \leq |\alpha|$ Por el lema anterior, tenemos que:
$[\beta] \leq 2^{\beta+1} - 2 < 2^{\beta+1} - 1 \leq 2^{|\alpha|} - 1 \leq [\alpha]$
$\Rightarrow [\beta] < [\alpha]$
$\therefore [\alpha] \neq [\beta]$.

Caso 2. $|\alpha| = |\beta|$. $\exists \rho, \tau, \pi$ tales que:
$\alpha = \tau a \rho$
$\beta = \pi b \rho$
$|\tau| = |\pi| = n$
$b + 1 \leq a$. Aquí, $b + 1$ se refiere al dígito sucesor de b.

$[\pi b] = [\pi] + [0^n b]$
Además, por el lema anterior sabemos que $[\pi] \leq 2^{n+1} - 2 < 2^{n+1} - 1$.
Así, $[\pi] + [0^n b] < (2^{n+1} - 1) + [0^n b] = [0^n(b + 1)] \leq [0^n a]$.

Con esto, podemos ver que

$[\beta] = [\pi b \rho] = [\pi b] + [0^{n+1}\rho]$
y por el argumento anterior
$< [0^n a] + [0^{n+1}\rho] \leq [\tau a \rho] = [\alpha]$
$\Rightarrow [\beta] < [\alpha]$
$\therefore [\alpha] \neq \beta$

□

Veamos ahora unos ejemplos de números binarios sesgados en su representación canónica:

- $10 \approx 011$
- $11 \approx 111$
- $12 \approx 211$
- $13 \approx 021$
- $14 \approx 002$

Discutimos en seguida la implementación de las funciones de incremento y decremento, trabajando únicamente con representaciones canónicas. Cada dígito tiene peso $2^{i+1} - 1$ y notemos que $1 + 2(2^{i+1} - 1) = 2^{i+2} - 1$. A partir de esto, podemos ver que incrementar un número binario sesgado cuyo primer

dígito distinto de cero es 2 se puede hacer fácilmente sustituyendo el 2 por 0 e incrementar el siguiente dígito de 0 a 1 o de 1 a 2 según corresponda. Incrementar un número binario sesgado que no contenga un 2 es más sencillo: simplemente hay que incrementar el primer dígito de 0 a 1 o de 1 a 2. En ambos casos, el número resultante está en su forma canónica y asumiendo que podemos encontrar el primer dígito distinto de cero en $O(1)$ ambos casos se ejecutan en $O(1)$.

Si usáramos una implementación densa, el primer dígito distinto de cero no podría encontrarse en $O(1)$, por lo que elegimos una implementación dispersa. Nuevamente representaremos un número binario sesgado mediante una lista de enteros, cuyos elementos serán los pesos. Veamos los ejemplos anteriores con esta implementación.

- $[3, 7] \approx 10$
- $[1, 3, 7] \approx 11$
- $[1, 1, 3, 7] \approx 12$
- $[3, 3, 7] \approx 13$
- $[7, 7] \approx 14$

En general, definimos a los naturales como una lista de enteros, pensando que los enteros en la lista deben ser de la forma requerida, es decir, de la forma $2^{i+1} - 1$.

```
data Nat = [Int]
```

Así definimos la función sucesor como sigue:

```
succ :: Nat -> Nat
succ w1:w2:ws | w1 == w2 = (1+w1+w2):ws
              | otherwise = 1:w1:w2:ws
succ ws = 1:ws
```

El primer caso verifica si los primeros dos pesos son iguales; si es así los combina para formar un peso del siguiente rango, en otro caso solo añade 1 al principio. El segundo resuelve el caso de la lista vacía o un solo elemento. Claramente ambos casos se ejecutan en $O(1)$.

Para decrementar tenemos dos casos: Si el primer peso de la lista es 1, el decremento es trivial quitándolo. En otro caso, tendremos un elemento de la forma $2^{i+1} - 1$. Necesitamos obtener $2^{i+1} - 2 = 2(2^i - 1)$. Y ya que, $\lfloor \frac{2^{i+1}-1}{2} \rfloor = 2^i - 1$, agregaremos 2 veces el resultado de dividir entre 2 el primer peso de la lista.

```
pred :: Nat -> Nat
pred (1:ws) = ws
pred (w: ws) = (w `div` 2) : (w `div` 2) : ws
```

Ahora discutamos como es que conocer los sistemas numéricos puede ayudarnos para construir una estructura que nos permita añadir y quitar elementos eficientemente.

2.5. Representaciones numéricas

Consideremos la representación de números naturales con un sistema numérico no posicional:

```
data Nat = Zero
  | Succ Nat
```

Lo único que recibe el constructor `Succ` es otro elemento de tipo `Nat` y el resultado de aplicar dicho constructor es obtener un número natural más grande en una unidad. ¿Por qué hemos nombrado `Succ`? Sólo por convención y dejar claro que este constructor nos da el sucesor de un natural. ¿Qué ocurriría si le cambiásemos el nombre? Por ejemplo de `Succ` a `Cons`, el `Zero` lo renombraramos como `Nil` y al tipo mismo lo renombraramos como `List`. Tendríamos el siguiente resultado:

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

El resultado se ve sospechosamente similar a la definición de listas. Lo único que hace falta es que el constructor `Cons` reciba como parámetro adicional un elemento de `a`. De esta manera obtenemos la definición usual de listas:

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

Hemos obtenido la definición de listas a partir de la definición de los números naturales. Analicemos que ocurre con las operaciones.

Incrementar un número natural se hace con el constructor `Succ` mientras que añadir un elemento se hace con el constructor `Cons`. Descartando el hecho de que `Cons` recibe un elemento de tipo `a`, efectivamente incrementar (añadir un elemento) a ambas estructuras se hace de manera análoga.

Por otro lado, para decrementar un número natural, basta quitar el primer constructor `Succ`. De manera análoga, quitar el primer elemento de una lista, se hace quitando el primer constructor `Cons` que se encuentra en la lista.

Las estructuras que tengan una analogía con alguna representación de los números naturales las llamaremos *representaciones numéricas*. La representación de los números naturales no es necesariamente la que vimos en el ejemplo anterior. Nuestro objetivo, después de darnos cuenta de esta analogía, será crear estructuras de datos que hereden las propiedades del sistema numérico en el que estén basadas. Es decir, se cumplirán los siguientes hechos:

- Una estructura que contenga n elementos, se modelará a través de la representación del número n en el sistema numérico.
- Agregar un elemento a una estructura de tamaño n será análogo a sumar 1 a la representación de n.
- Eliminar un elemento de una estructura de tamaño n corresponderá a restar 1 a la representación de n.

A lo largo de este documento desarrollamos distintas implementaciones de la estructura funcional conocida como *lista de acceso aleatorio*, la cual es una lista de árboles que sigue la idea desarrollada al final del capítulo anterior y cuya definición se basa en una representación numérica, es decir, las funciones de agregar y eliminar un elemento, se inspiran en las operaciones de sucesor y predecesor en el sistema numérico. El concepto de representación numérica será utilizado para implementar otras estructuras puramente funcionales, por ejemplo: Montículos binomiales. Para profundizar acerca representaciones numéricas, se puede consultar [6].

En este capítulo estudiamos los principales conceptos que requeriremos para definir una lista de acceso aleatorio. Aclaramos con precisión el concepto de lista, así como las estructuras llamadas árboles. Estos conceptos deberán estar muy presentes en el siguiente capítulo dado que las listas de acceso aleatorio se definirán usando ambos. Presentamos también el concepto de representaciones numéricas con el que modelaremos las listas de acceso aleatorio. También nos dimos cuenta de que al usar una implementación dispersa de un sistema numérico, el tipado no garantiza que la estructura sea válida, por ejemplo, el tipo de naturales se define simplemente como una lista de enteros; no hay garantía de que sus elementos sean pesos válidos, por ejemplo, potencias de 2. Este problema se hereda a las listas de acceso aleatorio y tendremos que buscarle una solución.

Capítulo 3

Listas de acceso aleatorio

En los capítulos anteriores hemos resuelto el problema del acceso aleatorio de manera teórica utilizando árboles binarios perfectos. Una vez que contamos con el concepto de representación numérica, podemos utilizarlo para obtener implementaciones de listas de acceso aleatorio. En adelante llamaremos `RAL` al tipo de listas de acceso aleatorio por sus siglas en inglés *Random Access Lists*.

Nuestro principal interés está en definir la siguiente biblioteca de operaciones para listas de acceso aleatorio:

```
-- Agregar un elemento al principio de una lista.
cons :: a -> RAL a -> RAL a

-- Separar una lista en cabeza y cola
uncons :: RAL a -> Maybe (a, RAL a)

-- Acceder a un elemento de la lista
lookup :: RAL a -> Int -> a

-- Actualizar un elemento de la lista
update :: RAL a -> Int -> a -> RAL a

-- Obtener una lista de acceso aleatorio a traves de una
    lista comun
fromList :: [a] -> RAL a

-- Obtener una lista comun a traves de una lista de
    acceso aleatorio
toList :: RAL a -> [a]
```

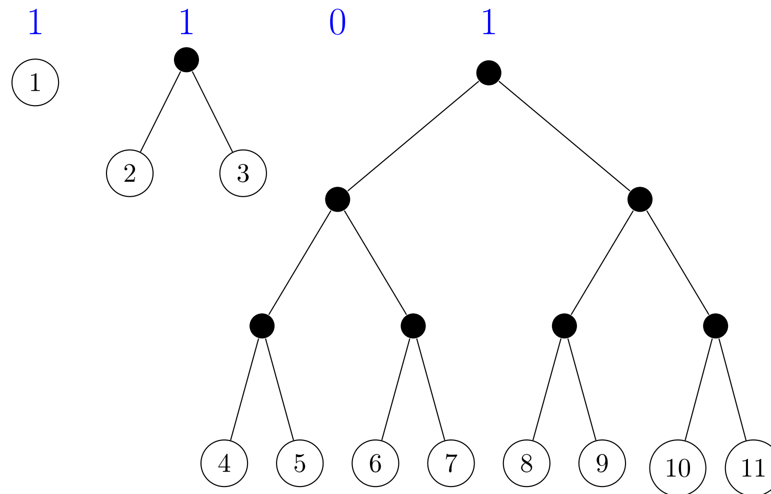
A lo largo de este documento, nos referiremos a las listas de acceso aleatorio como LAA, o con su nombre completo de manera indistinta.

3.1. Implementación binaria estándar

Como primera implementación utilizaremos una representación numérica basada en el sistema binario estándar $B = \langle D, W \rangle$ con $D = \{0, 1\}$ y $W =$

$\{1, 2, 4, 8 \dots\}$.

Como ya se dijo, una estructura que guarda n elementos se diseñará de acuerdo a la representación del número n en binario. Por ejemplo, si queremos guardar 11 elementos, debemos diseñar una estructura a partir del número binario 1101. Esto lo lograremos asociando a cada dígito d_i un árbol binario perfecto de rango i que solo guarda información en las hojas. Esto es conveniente puesto que el número de hojas en un árbol binario perfecto de rango i es precisamente 2^i. Siguiendo esta idea, la lista de los números del 1 al 11 se vería de la siguiente manera:



La definición formal en Haskell es:

```
data Digit a= One (Tree a)
  | Zero

data RAL a = [Digit a]
```

Los árboles se guardan en rango ascendente y el orden de los elementos es de izquierda a derecha, siendo la cabeza de la lista el hijo más a la izquierda del primer árbol. Notemos que el tamaño máximo de árboles en una lista de acceso aleatorio de tamaño n es $\lfloor \log(n + 1) \rfloor$ y el rango máximo de cualquier árbol es $\lfloor log(n) \rfloor$.

Como mencionamos anteriormente, al ser las listas de acceso aleatorio una representación numérica, las operaciones de añadir y quitar elementos serán análogas a las operaciones sucesor y predecesor de un número binario. Recordemos ambas:

```
inc [] = [One]
inc (Zero : ds) = One : ds
inc (One : ds) = Zero : (inc ds)
```

```
dec [One] = []
dec (One : ds) = Zero : ds
dec (Zero : ds) = One : (dec ds)
```

Para añadir un nuevo elemento al principio de la lista, la idea general es convertir el nuevo elemento en una hoja e insertar este nuevo árbol a la lista de árboles manteniendo los invariantes que mencionamos antes.

```
cons :: a -> RAL a -> RAL a
cons x l = consTree (Leaf x) l

consTree :: Tree a -> RAL a -> RAL a
consTree t [] = [One t]
consTree t (Zero : ts) = (One t) : ts
consTree t ((One t') : ts) = Zero : (consTree (link t t') ts)
```

Podemos observar nuevamente la analogía entre la función `inc` y la función `consTree`. La diferencia únicamente radica en el caso recursivo donde, además de la recursión, debemos ligar dos árboles binarios de rango r para poder mantener el invariante de tener árboles de rango i en la posición i. Para ello utilizamos la función `link`, cuya implementación es inmediata.

Para definir la función `uncons` requerimos primero una función más general `unconsTree` que dada una lista cuyo primer dígito tenga peso r, devolverá un par con un árbol de rango r y la lista resultante de eliminar ese árbol en la anterior, la cual debe mantener el invariante de ser una lista de árboles perfectos con rangos ascendentes. Esta función es en la que se basa la función decremento.

```
unconsTree :: RAL a -> Maybe (Tree a, RAL a)
unconsTree [] = None
unconsTree [One t] = Just (t, [])
unconsTree ((One t) : ts) = Just (t, Zero : ts)
unconsTree (Zero : ts) = let (Node t1 t2, ts') = unconsTree ts
  in (t1, (One t2) : ts')
```

Obsérvese que en el caso de tener una lista iniciando con `Zero`, el resultado del caso recursivo es siempre algo de la forma `(Node t1 t2, ts')`, i.e., la cabeza de una lista en el caso recursivo no puede ser una hoja. Ésto se debe a que ésta función considera como cabeza el primer árbol de la sucesión y debido al invariante de tener un árbol de rango i en la posición i, después del primer `Zero` únicamente existen árboles de rango $r > 0$.

La definición de `uncons` se obtiene de manera sencilla aplicando `unconsTree` para obtener un árbol de rango 0, es decir, una hoja, y la lista resultante de quitarla.

```
uncons :: RAL a -> Maybe (a, RAL a)
uncons ts = let (Leaf x, ts') = unconsTree ts in (x, ts')
```

Ambas operaciones ocupan una operación por dígito de la lista. Por lo tanto, su tiempo de ejecución es $O(\log(n))$ ya que existen a lo más $\lfloor \log(n+1) \rfloor$ árboles.

Las operaciones `head` y `tail` se obtienen como la primera y segunda proyección de la función `uncons`.

Las operaciones de acceso (*lookup* y *update*) no tienen una equivalencia con alguna operación binaria. No obstante, podemos implementarlas de manera eficiente aprovechando la estructura que hemos definido. Ambas operaciones son implementadas en dos pasos: primero buscamos el árbol necesario, posteriormente recorremos este árbol hasta llegar al elemento deseado.

```
lookup :: Int -> RAL a -> a
lookup i (Zero : ts) = lookup i ts
lookup i ((One t) : ts)
  | i < size t = lookupTree i t
  | otherwise = lookup (i - size t) ts

lookupTree :: Int -> Tree a -> a
lookupTree 0 (Leaf x) = x
lookupTree i (Node t1 t2)
  | i < (size t) `div` 2 = lookupTree i t1
  | otherwise = lookupTree (i - ((size t) `div` 2)) t2
```

Se observa que la función `lookupTree` realiza una búsqueda binaria usual sobre los índices de la lista. La función `update` se hace de manera análoga.

```
update :: Int -> a -> RAL a -> RAL a
update i y (Zero : ts) = update i y ts
update i y ((One t) : ts)
  | i < size t = (One (updateTree i y t)) : ts
  | otherwise = (One t) : (update (i - size t) y ts)

updateTree :: Int -> a -> Tree a -> Tree a
updateTree 0 y (Leaf x) = Leaf y
updateTree i y (Node t1 t2)
  | i < (size t) `div` 2 = Node (updateTree i y t1) t2
  | otherwise = Node t1 (updateTree (i - ((size t) `div` 2)) y t2)
```

Ambas operaciones recorren los dígitos hasta encontrar el árbol correcto, lo cual toma $O(\log n)$ y a lo más $O(\log n)$ para recorrer el árbol resultando en un total de $O(\log n)$ en el peor caso.

Finalmente, las funciones para convertir una lista convencional en una lista de acceso aleatorio y viceversa, las implementaremos de manera similar en todas las implementaciones vistas a lo largo de este documento. La primera transformación utiliza la operación de plegado sustituyendo el constructor (:) por el constructor `cons`, es decir, nuestra implementación es:

```
fromList :: [a] -> RAL a
fromList = foldr cons []
```

Hemos mencionado que las listas de acceso aleatorio guardan una analogía con el sistema numérico binario. La función `fromList` no queda fuera de esta analogía dado que su implementación se basa en llamadas sucesivas a la función `cons` que añade un elemento a la lista, análogamente a sumar 1 a un número binario. La función `toList` tampoco será la excepción, aunque la analogía es distinta. Dicha función es esencialmente la función de conversión del sistema binario al sistema unario. Puesto que, como ya mencionamos anteriormente, las

listas convencionales se basan en el sistema numérico unario, donde el dígito 1 es representado con el constructor CONS y el dígito 0 con el constructor NIL, restringiendo el uso del 0 únicamente para representar al 0 mismo.

Empecemos analizando como transformar un número binario en unario. Por ejemplo, si consideramos el número binario 0101, que representa al número decimal 10, ¿cómo obtenemos exactamente 10 dígitos 1? Sabemos que la primera aparición del 1, en el ejemplo, tiene peso $2^1 = 2$ y su segunda aparición tiene peso $2^3 = 8$, sumando ambos pesos obtenemos el número de dígitos 1 que requerimos. La implementación en HASKELL se torna un poco complicada porque hay que saber el peso de cada dígito 1 dentro de la sucesión de dígitos. Para ello, a la función de conversión agregamos un parámetro adicional que indicará el peso actual que representa cada dígito. Añadimos también la función `addSuccs` que será la encargada de añadir los constructores `Succ` necesarios.

```
data BinaryDigit = Zero | One
type Binary = [BinaryDigit]
data Unary = UZero | Succ (Unary)

toUnary :: Binary -> Int -> Unary
toUnary [] _ = UZero
toUnary (Zero:ts) weight = toUnary ts (weight*2)
toUnary (One:ts) weight = addSuccs weight (toUnary ts (weight*2))

addSuccs :: Int -> Unary -> Unary
addSuccs 0 u = u
addSuccs (n+1) u = Succ (addSuccs n u)
```

Aunque esta implementación se complicó más de lo deseado, nos daremos cuenta que siguiendo esta idea, la implementación de la conversión de listas de acceso aleatorio a listas comunes es mucho más sencilla. ¿La razón? Cada dígito sabe su peso a través de su árbol asociado. La analogía con nuestra función anterior es que cada dígito con peso 2^i debe agregar exactamente 2^i elementos a la estructura resultado. Esto es inmediato ya que cada árbol tiene 2^i elementos, que son los que agregaremos a la lista resultante.

Analicemos la función `toUnary`. Si recibe una lista vacía, es decir, una sucesión vacía de dígitos, devuelve `ZeroU`. En nuestro objetivo final, este caso se traduce en convertir una sucesión vacía de dígitos en una lista común. Esa lista común resultado es la lista vacía `Nil`, que en HASKELL es representada como [].

El siguiente caso es convertir una sucesión de dígitos que inicia con `Zero`. En este caso, la función `toUnary` aplica la recursión sobre la cola pero con el doble del peso. Este último detalle se debe a que debemos multiplicar por 2 el resultado, esto es para recordar cuantos dígitos hemos recorrido y así saber qué peso tienen los dígitos en la llamada recursiva. En nuestro objetivo final, este detalle no es necesario dado que se puede saber el peso de cada dígito `One` a través de su árbol asociado.

El último caso es transformar una sucesión de dígitos que inicia con `One`. Observemos que la función `toUnary` hace uso de la función `addSuccs` para añadir *weight* constructores `Succ`. Para nuestro objetivo final necesitaremos una función que añada *weight* constructores `Cons`. ¿Cuál es esta operación? Esta

operación será la concatenación. Se le concatenará una lista de *weight* elementos al resultado de aplicar recursión al resto de la sucesión. Para ello añadiremos los elementos de un árbol a la estructura resultante, *aplanando* el árbol asociado al dígito `One` que estamos revisando. El concepto de *aplanado* se refiere a convertir el árbol en una lista que contiene sus elementos en el mismo orden. Así la implementación será:

```
flat :: Tree a -> [a]
flat (Leaf x) = [x]
flat (Node t1 t2) = flat t1 ++ flat t2

toList :: RAL a -> [a]
toList [] = []
toList (Zero : ts) = toList ts
toList ((One t) : ts) = flat t ++ toList ts
```

Se observa que en comparación con la conversión anterior, en este caso, la definición se simplifica considerablemente puesto que el parámetro de peso y la función que agrega sucesores ya no son necesarias.

A lo largo de este documento, las implementaciones de estas dos funciones podrían presentar ciertas variaciones para adaptarse a las distintas versiones de las LAA que presentaremos. No obstante, la idea general será la misma.

Antes de continuar, presentaremos brevemente otra manera de construir una lista de acceso aleatorio de tamaño n. Esto nos será útil cuando queramos implementar arreglos de tamaño específico.

En la función `fromList` se añade elemento a elemento. Esto provoca que se haga una sucesión de llamadas a `cons`. ¿Podemos construir una LAA de manera más eficiente? Por supuesto, podemos reducir el problema de la construcción de la lista de tamaño n al problema de obtener la representación de n en el sistema binario. El proceso es como sigue:

- Si $n = 0$, se ha terminado el proceso.
- Si $n \neq 0$, se obtiene el residuo de la división $\frac{n}{2}$ y se coloca como el dígito menos significativo del resultado. El resto del resultado será la conversión del número $\lfloor \frac{n}{2} \rfloor$ al sistema numérico binario.

Con este proceso podemos construir fácilmente una lista de acceso aleatorio de tamaño n, siempre y cuando, los elementos a guardar sean el mismo. La implementación es:

```
replicate :: Int -> Tree a -> RAL a
replicate 0 _ = []
replicate n t = if n `mod` 2 == 1 then
                (One t):(replicate (n `div` 2) (Node t t))
                else
                Zero:(replicate (n `div` 2) (Node t t))
```

Notemos que el caso recursivo se hace con el elemento `Node t t`. Dado que la cola de la lista debe contener árboles de un rango más alto. Es por esta razón

que se vuelve complicado implementar la función `fromList` con este método ya que a pesar de que conocemos que forma tendrá la estructura final, no sabemos exactamente que elementos de la lista original deben guardarse en que árboles.

Aunque la función `replicate` puede ser útil en algunos casos, no es el objetivo que estamos buscando. Queremos implementar la función `fromList`. El problema con `replicate` es que recibe un número y un elemento a replicar. Podríamos decir que `replicate` convierte un número n en su representación decimal a un número en un sistema numérico binario. Este proceso es sencillo y su implementación es muy sencilla, es por ello que nos gustaría tener un proceso similar para convertir una lista común (una representación numérica en un sistema numérico unario) en una lista de acceso aleatorio (una representación numérica en un sistema numérico binario). La conversión del sistema numérico decimal al binario se usan las funciones `mod` y `div`. Estas funciones están implementadas para números enteros en un sistema numérico decimal. Su implementación en un sistema numérico unario también es simple. Para nuestro objetivo, queremos implementarlas directamente en la representación numérica, es decir, queremos implementarlas sobre listas comunes. Lo primero que debemos hacer es definir funciones análogas a `mod` y `div` pero que funcionen sobre listas convencionales.

La implementación de estas operaciones se facilitará si las juntamos en una única función, a la que llamaremos `modDiv2`. Esta función devolverá un par, cuyo primer elemento es el resultado de la operación `mod` y el segundo el resultado de la función `div`. Lo primero que definiremos es: ¿qué significa la división de una lista entre 2? Esto significará convertir una lista de longitud n en una lista de longitud $\lfloor \frac{n}{2} \rfloor$ pero que contenga exactamente los mismos elementos. Dividir una lista con esta restricción, es imposible en este punto de nuestro análisis, puesto que no hay manera de tomar una lista de elementos de tipo `a` y devolver una lista con los mismos elementos pero de longitud menor preservando el tipo. Es por ello que restringiremos esta función a recibir únicamente listas de árboles. Así, si tenemos una lista de árboles de `a`, al dividir entre 2, la lista resultante tendrá los mismos elementos de tipo `a` pero contenidos en árboles más grandes. Es decir, para reducir la longitud de la lista a la mitad, pegaremos los árboles de la lista por pares. Esto nos lleva a un problema: ¿qué pasa si la longitud de la lista es impar? habría un árbol que no podría ser pegado con otro, este árbol lo descartaremos del resultado de la función `div`, y ahora es cuando la función `mod` debe hacer acto de presencia. Este árbol sobrante significa que la lista tenía longitud impar, es decir, el resultado de la operación `mod` no es cero sino dicho árbol sobrante. Así, la función `mod` tiene dos posibles resultados: un dígito `Zero` o un dígito `One` con un árbol asociado. Convenimos también que cuando la lista tenga longitud impar, el árbol asociado al resultado de `mod` será el primer árbol de la lista.

La implementación de `modDiv2` se hará recursivamente con la idea intuitiva siguiente: si la lista de árboles es vacía, el módulo es `Zero` y la lista resultante de `div` es vacía. En otro caso, tenemos dos posibilidades. Si el módulo de la cola es `Zero`, entonces el resultado final es módulo `One` con la cabeza de la lista original como árbol asociado y como lista resultante de la división, la misma

del resultado recursivo. Si el módulo de la cola es `One` con un árbol asociado `t`, entonces el resultado final es módulo `Zero`, y como lista resultante de la división, la misma del caso recursivo, añadiendo el árbol resultante de pegar la cabeza de la lista original y `t`, el árbol asociado al dígito `One` devuelto por el caso recursivo.

Para su implementación en HASKELL, debemos hacer uso de otros constructores para los dígitos `Zero` y `One`, ya que estos están reservados para las listas de acceso aleatorio. Así, la implementación final de `modDiv2` es:

```
data DigitsAux a= ZeroAux | OneAux a

modDiv2 :: [Tree a] -> (DigitsAux (Tree a), [Tree a])
modDiv2 [] = (ZeroAux, [])
modDiv2 (t:xs) = case modDiv2 xs of
                 (ZeroAux, q) -> (One t, q)
                 (OneAux t', q) -> (Zero, (Node t t'):q)
```

Haciendo uso de esta función auxiliar, podemos implementar la función `fromList` de inmediato:

```
fromList :: [a] -> RAL a
fromList l = fromListAux (map (\x->Leaf x) l)

fromListAux :: [Tree a] -> RAL a
fromListAux [] = []
fromListAux l = case modDiv2 l of
              (ZeroAux, q) = Zero : (fromList q)
              (OneAux t, q) = (One t) : (fromList q)
```

Esta implementación, además de ser más elegante, deja más clara la analogía con la conversión entre sistemas numéricos.

Las listas de acceso aleatorio que hemos presentado resuelven parcialmente nuestro problema. Ya tenemos el acceso aleatorio pero existen algunas desventajas. Como por ejemplo, la función `cons` se ejecuta en $O(\log n)$ y en las listas convencionales se ejecuta en $O(n)$. Es por ello que en las siguientes secciones presentaremos un par de versiones alternativas a ésta.

3.2. Implementación sin ceros

La primera desventaja que observamos en la implementación vista en la sección anterior es que las funciones *cons* y *uncons* se ejecutan en $O(\log n)$ en vez de $O(1)$.

Obtener la cabeza de una lista debe de hacerse a través de la función *uncons* que obtiene el primer elemento y reconstruye la lista quitando ese elemento. Esto reduce las líneas de código y funciones a implementar pero desperdicia tiempo de ejecución construyendo una lista que será descartada si sólo se busca obtener la cabeza de la lista. Para mayor eficiencia, implementemos primero la función *head*. Un caso de *head* que corre en $O(1)$ es cuando la lista comienza con el dígito 1.

```
head ( (One (Leaf x)): _)= x
```

Al observar esto, nos gustaría que el primer dígito de la lista nunca fuese 0. Actualmente nuestro conjunto de dígitos es $\{0, 1\}$. Sin embargo, esto no es una regla. Podemos cambiar de representación de números binarios como mejor nos convenga. Utilicemos una representación sin ceros; esto es, utilizaremos un conjunto de dígitos que no contenga al 0 y el mismo conjunto de pesos. Utilizaremos el conjunto de dígitos $\{1, 2\}$. Así el número decimal 16 se representa como 2111 en vez de 00001. Adecuar nuestras listas al cambio de dígitos se hace de una manera fácil. Al dígito 1 le seguimos asociando un árbol y al dígito 2 le asociamos un par de árboles. De hecho, podríamos asociarle al dígito 2 un único árbol de rango $i + 1$ donde i es la posición del dígito pero por claridad le asociaremos 2 árboles de rango i.

La definición del tipo cambia de esta manera:

```
data Digit a= One (Tree a)
  | Two (Tree a, Tree a)

data RAL a = [Digit a]
```

La implementación final de `head` es:

```
head :: RAL a -> a
head ( (One (Leaf x)) : _) = x
head ( (Two (Leaf x, _)) : _) = x
```

Que claramente se ejecuta en $O(1)$ en todos los casos. Ahora, la definición de la función `tail` es:

```
tail :: RAL a -> RAL a
tail ( (Two (t1, t2)) : xs) = (One t2) : xs
tail ( (One _) : xs) = let (Node t1 t2) = head xs in (Two (t1, t2))
     : (tail xs)
```

Esta implementación de `tail`, además de que mantiene el mismo tiempo de ejecución que la versión anterior, conlleva más problemas. Esta implementación claramente falla cuando la lista no contiene ningún dígito 2, puesto que, al final de la ejecución intentará obtener la cola de la lista vacía, la cual dijimos que no está definida. Resolver este inconveniente nos llevaría a manejar más casos o definir la cola de la lista vacía. Eso nos resulta poco conveniente, dado que realmente no ganamos demasiado con este cambio. Es por ello que mostraremos otra versión de las listas de acceso aleatorio.

3.3. Implementación binaria sesgada

Aunque resolvimos el problema del acceso a la cabeza de una lista, aún tenemos el problema de que las funciones de incremento y decremento se ejecutan en $O(\log n)$. Este problema lo resolverá el sistema numérico binario sesgado descrito anteriormente.

Hemos mostrado las operaciones para incrementar y decrementar números binarios sesgados. Nuestro objetivo actual es utilizar esta representación para

diseñar listas de acceso aleatorio. Dado que utilizaremos una implementación dispersa, definiremos las listas de acceso aleatorio como una lista de pares de enteros y árboles, los cuales pensamos como árboles binarios perfectos, de tal forma que el entero representa el rango del árbol. Más aún, los árboles serán guardados en rango ascendente y convenimos en que sólo los primeros dos árboles pueden ser del mismo rango. Esto corresponde a la propiedad del sistema binario segado acerca de que el dígito 2 sólo puede aparecer una vez y debe ser el primer dígito distinto de cero. Sabemos que los árboles perfectos tienen 2^i hojas pero el sistema sesgado requiere pesos de la forma $2^{i+1} - 1$. Este problema se resuelve utilizando árboles perfectos con información en todos sus nodos, puesto que ya vimos que un árbol perfecto de rango i tiene $2^{i+1} - 1$ nodos.

En la sección anterior, también resolvimos el problema de acceder a la cabeza en $O(\log n)$ reduciéndolo a $O(1)$. Esta solución se vería estropeada si la cabeza sigue siendo el elemento más a la izquierda del árbol. Ahora guardamos datos en los nodos internos, por lo que optaremos por guardar la información del árbol en preorden en cada árbol. Así la cabeza de la lista será la raíz del primer árbol, recuperándose el tiempo de ejecución $O(1)$.

Así la nueva definición de nuestras LAA es:

```
data Tree a = Leaf a
  | Node a (Tree a) (Tree a)

data RAL a = [(Int, Tree a)]
```

La operacion para agregar un elemento a la cabeza es:

```
cons :: a -> RAL a -> RAL a
cons x ts@((w1, t1):(w2, t2): rest)
  | w1 == w2 = (1+w1+w2, Node x t1 t2) : rest
  | otherwise = (1, Leaf x) : ts
cons x ts = (1, Leaf x) : ts
```

La implementación se explica a si misma recordando la definición de la función `succ` en la página 17.

Las operaciones *head* y *tail* se implementan fácilmente puesto que la cabeza de la lista siempre está en la raíz del primer árbol. La implementación es:

```
head :: RAL a -> a
head ((_, Leaf x) : _) = x
head ((_, Node x _ _) : _) = x
```

Por otra parte, la operación *tail* también se implementa fácilmente siguiendo la correspondiente definición de la función `pred` de la página 17.

```
tail :: RAL a -> RAL a
tail ((_, Leaf _) : ts) = ts
tail ((w, Node _ t1 t2) : ts) = (w `div` 2, t1) : (w `div` 2, t2) :
     ts
```

Adecuar las operaciones de búsqueda y actualización es casi inmediato. Éstas recibirán también el tamaño del árbol para saber cuantos elementos tenemos en el árbol izquierdo y en el derecho.

```
lookup :: Int -> RAL a -> a
lookup i ((w, t) : ts)
  | i < w = lookupTree w i t
  | otherwise = lookup (i-w) ts

lookupTree :: Int -> Int -> Tree a -> a
lookupTree 1 0 (Leaf x) = x
lookupTree _ 0 (Node x _ _) = x
lookupTree w i (Node _ t1 t2)
  | i < w `div` 2 = lookupTree (w `div` 2) (i-1) t1
  | otherwise = lookupTree (w `div` 2) (i - 1 - (w `div` 2)) t2
```

En el caso recursivo, buscamos el elemento $i-1$ dado que ya descartamos la raíz del árbol original, a saber el elemento x. De manera similar, en la última línea ya hemos recorrido $1+\lfloor \frac{w}{2} \rfloor$ elementos, que son los correspondientes a `t1` y a `x`, por lo que en el caso recursivo los restamos a `i`. La operación de actualización se define de manera análoga y su implementación se muestra en el apéndice este documento.

Es claro que las operaciones *cons*, *head* y *tail* se ejecutan en $O(1)$ y las operaciones *lookup* y *update* se ejecutan en $O(\log n)$

Las implementaciones vistas hasta ahora fueron realizadas fácilmente. Bastó razonar sobre la estructura siguiendo la analogía con el sistema numérico correspondiente obteniendo así la implementación deseada. Sin embargo, hasta ahora no nos hemos puesto a pensar sobre la pregunta ¿El tipo definido `RAL` realmente corresponde a la especificación dada en la teoría? La respuesta a esta pregunta es claramente "no".

En la teoría las listas de acceso aleatorio son sucesiones de dígitos donde cada dígito d distinto de cero en la posición i tiene asociados d árboles binarios perfectos de rango i. Sin embargo, la implementación no obliga que los árboles utilizados sean en realidad árboles binarios perfectos. Puesto que el tipo `Tree a` genera árboles binarios cualesquiera, por ejemplo:

```
Node (Node (Leaf 1) (Leaf 2)) (Leaf 5)
```

Es un árbol de tipo de `Tree Int` pero no es un árbol binario perfecto. Más aún, consideremos el siguiente elemento del tipo `RAL Int`

```
[One (Leaf 5), One (Leaf 4), One (Leaf 3)]
```

¡También es una LAA válida bajo la definición del tipo! No obstante, está lejos de ser una lista de acceso aleatorio válida pues todos sus árboles son del mismo rango.

En conclusión, a pesar de que nuestra implementación fue bastante directa, las listas de acceso aleatorio son un subconjunto del tipo `RAL`. Una manera de resolver este problema es implementar una función `check` como lo hicimos en la implementación dispersa de los números binarios. No obstante, habría que utilizar esta función de verificación en todas las funciones que queramos definir. En su lugar, en el siguiente capítulo discutiremos la posibilidad de forzar las invariantes del tipo desde su propia definición utilizando conceptos avanzados de la programación funcional.

Hemos definido las listas de acceso aleatorio y nos hemos dado cuenta que resultan ser una mejor alternativa que las listas comunes cuando de buscar y actualizar elementos aleatorios se trata. La implementación de las operaciones siguió la analogía con los sistemas numéricos, mostrando así, el beneficio de utilizar conceptos matemáticos para motivos prácticos de programación.

A pesar de las ventajas de las listas de acceso aleatorio, hay un problema latente que venimos arrastrando desde el capítulo anterior. El tipado no nos da ningún tipo de garantía sobre la estructura, por lo que el usuario podría alimentar nuestras funciones con entradas inválidas y el resultado sería erróneo. En el siguiente capítulo daremos la solución a este problema mostrando tipos que nos garantizarán invariantes de la estructura. Adaptaremos también nuestras listas de acceso aleatorio a este tipado.

Capítulo 4

Tipos de datos anidados

Las implementaciones de las listas de acceso aleatorio vistas hasta ahora nos han permitido solucionar los inconvenientes con respecto al tiempo de ejecución y espacio de almacenamiento. Sin embargo, aun existe una gran desventaja. Nunca se garantiza que las estructuras cumplan los invariantes que hemos dado. Por ejemplo:

```
Node (Node (Leaf 1) Empty) (Leaf 5)
```

Es un árbol válido bajo la definición del tipo. Sin embargo, no es un árbol binario perfecto, que son los árboles que requerimos para las listas de acceso aleatorio. De esta manera es posible construir elementos del tipo `RAL a` que no sean listas de acceso aleatorio de acuerdo a la definición.

Este problema se puede solucionar, como hemos discutido antes, con una función de verificación que decida si los árboles binarios involucrados en una lista de acceso aleatorio son realmente árboles perfectos. Esta función es:

```
hg :: Tree a -> Int
hg (Leaf x) = 0
hg (Node t1 t2) = 1 + max (hg t1) (hg t2)

perfect :: Tree a -> Bool
perfect (Leaf x) = True
perfect (Node t1 t2) = perfect t1 &&
                       perfect t2
                       && hg t1 == hg t2
```

Sin embargo, ya se dijo que el uso de esta función es fuente de ineficiencia por lo que nos gustaría garantizar los invariantes de una estructura de datos de manera estática, por ejemplo, quisiéramos que todos los elementos del tipo `RAL a` sean listas de acceso aleatorio válidas. Para ello utilizaremos conceptos avanzados de la programación funcional. En particular, el concepto de tipos de datos anidados.

Un tipo de datos anidado es un tipo de datos recursivo cuyo parámetro es distinto al original en la llamada recursiva. Por ejemplo, consideremos la

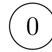
siguiente definición de árboles perfectos. Un árbol perfecto con etiquetas en `a` es una hoja etiquetada `Zero a` o bien un árbol perfecto con etiquetas en `(a,a)`. Esta definición corresponde a un tipo anidado, a saber:

```
data Perfect a =  Zero a
                | Succ (Perfect (a,a))
```

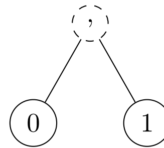
Podemos observar que estamos definiendo el tipo `Perfect` sobre elementos de `a`. Curiosamente del lado derecho, el constructor `Succ` recibe un elemento de tipo `Perfect (a,a)`, por lo que la recursión no es uniforme.

La pregunta natural es ¿y eso para qué sirve? A simple vista la respuesta no es tan clara, así que analicemos algunos elementos de tipo `Perfect Int`:

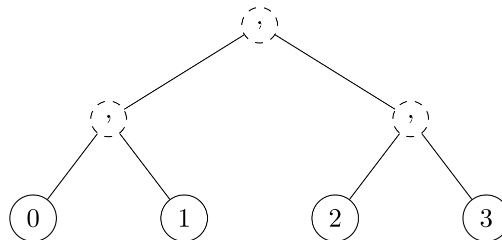
- `Zero 0`



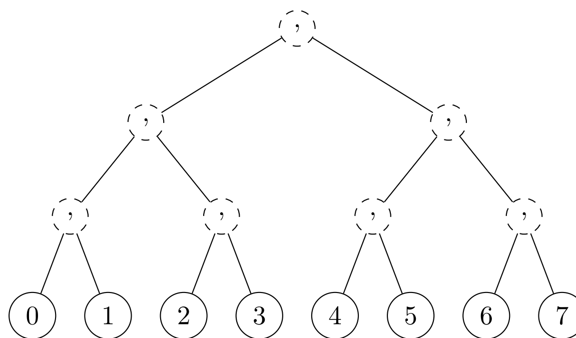
- `Succ (Zero (0,1))`



- `Succ (Succ (Zero ((0,1),(2,3))))`



- `Succ ( Succ ( Succ ( Zero ((0,1),(2,3), ((4,5),(6,7))) )))`



Hemos elegido poner una coma en los nodos internos de la representación gráfica de estos árboles para indicar la formación de tuplas de acuerdo a la siguiente transformación de un árbol binario común, es decir, de un elemento `Tree a`, a un elemento del tipo `Perfect a`:

- Una hoja `Leaf x` se convierte en una hoja `Zero x`.
- Cada árbol común de la forma `Node (Leaf x) (Leaf y)` se convierte en una hoja de la forma `Zero (x,y)`.
- Se lleva la cuenta de cuantas veces se aplica la transformación. Esta cuenta será el número de constructores `Succ` que utilizaremos, ya que, un elemento de la forma `Zero (x, y)` tiene tipo `Perfect (a,a)` y queremos obtener un elemento de tipo `a`. Curiosamente, el número de `Succ` que utilicemos, será también el rango del árbol.

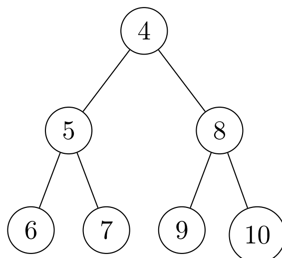
Si se aplica esta transformacion recursivamente, se puede transformar cualquier árbol binario perfecto común en un árbol de tipo `Perfect`. Es claro que si el árbol binario común no era perfecto, la transformación fallará, por lo que efectivamente, si t es de tipo `Perfect a`, entonces necesariamente t corresponde un árbol binario perfecto. Con lo cual se ha logrado nuestro objetivo de asegurar de manera estática el invariante de la definición de árboles perfectos.

Con esto hemos logrado definir árboles perfectos con datos en las hojas, pero ¿qué ocurre si queremos representar árboles con datos en los nodos? Una posible solución es añadir un tipo `Node` que representará los nodos del árbol:

```
data Tree a k = Zero a
              | Succ (Tree (Node a k) k)

data Node a k = Node a k a
```

El tipo `Node a k` construye árboles binarios cuya raíz está etiquetada por `k` y sus subárboles son de tipo `a` (estamos pensando que el tipo a es un tipo contenedor de elementos de k, i.e., el tipo `a` representa una estructura que puede guardar elementos de tipo `k`, por ejemplo, un árbol). En particular, el tipo `Node () k` construye hojas con elementos de k, puesto que en este caso, los subárboles siempre son vacíos ya que son elementos del tipo `()`. Así podemos construir árboles más complejos, por ejemplo:



Obsérvese que si nos paramos en cualquier nodo de este árbol, el subárbol cuya raíz es dicho nodo se puede ver como instancia del tipo `Node a k`. Por ejemplo, el subárbol cuya raíz es 5, puede verse como un elemento del tipo `Node (Node () Int) Int`. Siguiendo este proceso de identificación de subárboles con elementos de algún tipo de la forma `Node a k` y contando el número de veces que se hizo esta identificación, mediante el uso de los constructores `Zero` y `Succ`, se obtiene un elemento del tipo anidado `Tree a k`. Los constructores `Zero` y `Succ` nos ayudarán a obtener fácilmente el rango del árbol. Por ejemplo, el árbol representado en la figura anterior se implementa como:

```
Succ(
  Succ(
    Succ(
      Zero(
        Node(
          (Node
            (Node ()  6 ())
          5
            (Node ()  7 ())
          )
        4
          (Node
            (Node ()  9 ())
          8
            (Node () 10 ())
          )
        )
      )
    )
  )
)
```

Ahora que contamos con tipos de datos anidados para árboles binarios perfectos, ya es posible desarrollar las implementaciones de listas de acceso aleatorio, lo cual hacemos a continuación.

4.1. LAA con tipos de datos anidados

En la sección anterior construimos árboles binarios perfectos con datos en las hojas y en los nodos. Empezaremos adaptando nuestra primera implementación que utiliza árboles binarios perfectos con datos en las hojas. ¿Cómo adaptamos esta representación para representar listas de acceso aleatorio?

Veamos algunos ejemplos de manera intuitiva, apegándonos a la representación numérica, es decir, utilizando constructores `Zero` y One.

- La lista vacía se representa con `Nil`.
- La lista $[1]$ se representa con `One 1 Nil`.
- La lista $[1, 2]$ se representa como `Zero (One (1,2) Nil)`.
- La lista $[1, 2, 3]$ se representa como `One 1 (One (2,3) Nil)`.

- La lista $[1, 2, 3, 4]$ se representa como `Zero (Zero (One ((1,2),(3,4)) Nil))`.
- La lista $[1, 2, 3, 4, 5]$ se representa como `One 1 (Zero (One ((2,3),(4,5)) Nil))`.

Se observa que el constructor `One` juega el papel del constructor `Cons`, es decir, agrega un elemento a la lista. Mientras que el constructor `Zero` encapsula listas de pares de `a` para considerarlas listas de elementos de `a`. La definición de listas de acceso aleatorio es la siguiente:

```
data RAL a = Nil
           | Zero (RAL (a,a))
           | One a (RAL (a,a))
```

Obsérvese que el anidamiento forza a que el constructor `One` añada 2^k elementos cada vez que se utiliza, esto no causa restricción alguna en la longitud de la lista debido a que todo número tiene una descomposición binaria. Más aún, como se ve en los ejemplos anteriores, la composición de los constructores `Zero` y `One` genera la representación binaria de la longitud de la lista.

Puesto que la implementación es esencialmente su representación binaria, volvemos a utilizar un razonamiento aritmético para nuestra implementación.

Las operaciones `cons` y `uncons` son idénticas a la versión sin tipos de datos anidados, a excepción de que separar y unir árboles ahora sólo se resume en separar y crear tuplas. Por lo tanto, las funciones `head` y `tail` también son idénticas.

```
cons :: a -> RAL a -> RAL a
cons x Nil = One x Nil
cons x (Zero ps) = One x ps
cons x (One  y ps) = Zero (cons (x,y) ps)

uncons :: RAL a -> (a, RAL a)
uncons (One x Nil) = (x, Nil)
uncons (One x ps) = (x, Zero ps)
uncons (Zero ps) = (x, One y ps')
                   where ((x,y), ps') = uncons ps

head xs = x where (x, _) = uncons xs
tail xs = xs' where (_,xs') = uncons xs
```

Pero detengámonos un momento a analizar la función `cons`. Ésta recibe listas de acceso aleatorio sobre `a` y un elemento de `a`, pero observando la llamada recursiva, podemos observar que `cons` que está recibiendo algo de tipo `(a, a)` y una lista de acceso aleatorio sobre `(a, a)`, es decir, ¡la llamada recursiva no tiene la misma firma que la llamada original! Cuando esto ocurre, diremos que la recursión es *polimórfica* o *no uniforme*. Utilizar tipos de datos anidados implica que muchas funciones utilizarán este tipo de recursión, por lo que se requerirá un razonamiento más complicado para construir la implementación. En particular, toda función recursiva polimórfica requiere la declaración de su firma, puesto que en otro caso, el intérprete de HASKELL arrojará un error de unificación.

Las funciones `lookup` y `update` requieren un razonamiento diferente dado que la recursión ya no es uniforme.

Para obtener un elemento en una lista que empieza con `One x xs` regresamos `xs` i buscamos el elemento en la posición 0, o volvemos a aplicar la función sin considerar `x`, es decir, sobre la misma secuencia sustituyendo `One` con `Zero` y buscando el elemento $i-1$ dado que acabamos de quitar un elemento.

Ahora bien, si la secuencia empieza con `Zero` obtendremos el par en la posición $\lfloor \frac{i}{2} \rfloor$ y luego obtendremos el elemento correcto de ese par. Es fácil ver que todos los elementos con índice par están en los subárboles izquierdos. Si i es par, entonces devolvemos el primer elemento de la tupla. Si no, devolvemos el segundo.

```
lookup :: Integer -> RAL a -> a
lookup 0 (One x _) = x
lookup i (One _ xs) = lookup (i-1) (Zero xs)
lookup i (Zero xs) = let (x,y) = lookup (i `div` 2) xs
                 in if (i `mod` 2) == 0 then x else y
```

Finalmente, discutimos la operación para actualizar. Los casos cuando la lista empieza con el dígito `One` son sencillos: si el elemento a actualizar es el primero, simplemente cambiamos el elemento y dejamos la cola igual. Por otro lado, si requerimos actualizar el índice i procedemos igual que con `lookup`, actualizamos el índice $i-1$ de la lista sustituyendo el primer dígito `One` por `Zero` y a este resultado le añadimos el elemento que tenía anteriormente el dígito `One`.

Si la lista inicia con `Zero` tenemos un problema porque requerimos actualizar el par en la posición $\lfloor \frac{i}{2} \rfloor$ para lo cual necesitamos del otro elemento de la antigua tupla para construir la nueva. Para resolver esto, tenemos que hacer una llamada a la función `lookup` en la posición $\lfloor \frac{i}{2} \rfloor$, la cual devuelve un par que contiene al elemento original a actualizar. Este par se actualiza de manera adecuada en relación a la paridad del índice original y se utliza en la llamada recursiva a `update`. La implementación es como sigue:

```
update :: Integer -> a -> RAL a -> RAL a
update 0 e (One x xs) = One e xs
update i e (One x xs) = cons x (update (i-1) e (Zero xs))
update i e (Zero xs) =
  let
    (x,y) = lookup (i `div` 2) xs
    p = if (i `mod` 2) == 0 then (e,y) else (x,e)
  in
```

Para terminar esta implementación, hay que adaptar las funciones de conversión de una lista común a una lista de acceso aleatorio y viceversa. La primera se mantiene igual:

```
fromList :: [a] -> RAL a
fromList = foldr cons Nil
```

Mientras que la segunda es un poco más compleja que antes, pero la idea es similar, repetimos varias veces la operación `uncons` hasta llegar a la lista de acceso aleatorio vacía.

```
toList :: RAL a -> [a]
toList Nil = []
toList l = let (x, xs) = uncons l in x:(toList xs)
```

Esta no es la solución óptima pero es la más intuitiva. Para la versión mejorada de nuestras LAA con tipos de datos anidados, que discutiremos en la siguiente sección, mostraremos una versión mucho más eficiente para convertir a una lista común.

Con esto terminamos nuestra primera implementación con tipos anidados, la cual, utilizó una representación numérica binaria estándar. A continuación, desarrollamos la última implementación de este trabajo, utilizando la representación binaria sin ceros.

4.2. LAA de orden superior

La definición de árboles perfectos dada por el tipo anidado `Tree a k` discutida en la página 35 tiene una deficiencia importante, a saber, no existe relación alguna entre el tipo `k` y el tipo `a`. El hecho de que `a` sea un contenedor de elementos de `k` es de gran importancia pero no esta forzado en la definición del tipo, por lo que la implementación resultaría poco robusta. Resolvemos dicho inconveniente utilizando lo que se conoce como un *tipo anidado de orden superior*, que es un tipo donde la recursión se hace sobre constructores de tipos en vez de tipos como tal. Empecemos dando una versión de orden superior del tipo de árboles binarios perfectos con datos en las hojas:

```
data Perfect bush a = Zero (bush a)
                    | Succ (Perfect (Fork bush) a)

data Fork bush a = Fork (bush a) (bush a)
```

Como se observa, esta versión hace explícito que el primer argumento `bush` es un contenedor del segundo argumento `a`. Esto se nota al observer que utilizamos `bush a`, ya que esto forza a que `bush` tenga tipo $* \to *$, es decir, es un constructor de tipos. Más aún, el tipo `Fork` recibe como argumento el constructor de tipos `bush` (pensemos que `Fork` es un constructor de árboles) y un tipo `a` y devuelve un árbol obtenido pegando dos árboles del tipo `bush a`. Los elementos del tipo `Perfect bush a` son árboles construidos por `bush` encapsulados en el constructor `Zero`, o bien, árboles perfectos, donde el constructor `bush` se sustituyó por `Fork bush`, que es un constructor que pega dos árboles del tipo `bush a`.

A continuación utilizamos un tipo de datos similar a `Perfect` para implementar listas de acceso aleatorio, la cual se basa en la representación sin ceros, utilizando el sistema numérico binario con dígitos $\{1, 2\}$.

En la implementación anterior de las listas de acceso aleatorio sin ceros, discutida en la sección 3.2 asociamos dos árboles de rango i a cada dígito 2. No obstante, podemos observar que, en su lugar, podemos asociarle un único

árbol de rango $i+1$. Hecha esta observación podemos definir las listas de acceso aleatorio como sigue:

```
data Leaf a = Leaf a
data Fork bush a = Fork (bush a) (bush a)

data HRAL bush a =
       Nil
     | One (bush a) (HRAL (Fork bush) a)
     | Two (Fork bush a) (HRAL (Fork bush) a)
```

Utilizamos el tipo `HRAL` para dejar claro que éstas listas son LAA con tipo de datos de orden superior; adoptamos la `H` de las siglas en inglés de *Higher order nested data types*. Las listas de acceso aleatorio que utilizará el usuario final se definen a partir de este tipo como:

```
type RAL a = HRAL Leaf a
```

Empecemos a definir las funciones necesarias para la biblioteca de listas de acceso aleatorio. La función `cons` se hará utilizando una función auxiliar que llamaremos `incr` que dado un elemento de `Fork`^{i} `Leaf` y una lista de acceso aleatorio sobre `Fork`^{i} `Leaf` devuelve una lista de acceso aleatorio sobre `Fork`^{i} `Leaf`. Las definiciones correspondientes son:

```
cons :: a -> RAL a -> RAL a
cons a s = incr (Leaf a) s

incr :: bush a -> HRAL bush a -> HRAL bush a
incr b Nil = One b Nil
incr b1 (One b2 ds) = Two (Fork b1 b2) ds
incr b1 (Two b2 ds) = One b1 (incr b2 ds)
```

Nuevamente, la función `incr` está basada en la función de incremento en el sistema numérico. Lo que sí resulta conveniente analizar es que significa tener una lista de acceso aleatorio sobre `Fork`^{i} `Leaf`. Esto queda más claro al darnos cuenta que la función `incr` utiliza recursión polimórfica. La llamada recursiva del tercer caso tiene tipo `Fork bush a ->HRAL (Fork bush) a ->HRAL (Fork bush) a`, que es distinto al tipado declarado en la definición. Con esto podemos darnos cuenta que las listas de acceso aleatorio que utilizaremos comunmente, o las que usaría el usuario final, tienen tipo `HRAL Leaf a`. No obstante, al definir operaciones sobre estas listas, tendremos que trabajar con la cola de una lista con dicho tipo, y su cola tendrá tipo `HRAL (Fork Leaf) a` y a su vez su cola tendrá tipo `HRAL (Fork (Fork Leaf)) a`. A esto nos referiremos cuando digamos que tenemos listas de acceso aleatorio sobre `Fork`^{i} `Leaf`; esto será equivalente a tener una lista con tipo `HRAL (Fork`^{i} `Leaf) a`.

El algoritmo visto para decrementar un número binario presentado anteriormente es el siguiente:

```
dec :: Nat -> Nat
dec [One] = []
dec (One :: ds) = Two :: (dec ds)
dec (Two :: ds) = One :: ds
```

[0]`Fork`^{i} representa aplicar el constructor `Fork` i veces

Pero ahora mostraremos otra versión más elegante. Los primeros dos casos pueden ser simplificados si nos permitimos tener un único 0 de manera temporal y aplicando las siguientes reglas para quitarlo:

- Eliminar el 0 de la secuencia cuyo único elemento es él, resulta en la secuencia vacía.
- Si la secuencia tiene más de dos elementos, nos fijamos en el dígito siguiente al único 0. Supongamos que el 0 está en la posición i y procedemos como sigue:
  - Si la posición $i+1$ tiene al dígito 1, éste tiene asociado el peso 2^{i+1} y observemos que $2^{i+1} = 2(2^i)$. Dicha está observación, quitamos el dígito 0, en la posición i, sustituyéndolo por un dígito 2. Además para preservar la suma de los pesos, el 1 en la posición $i+1$ se cambia por 0.
  - Si la posición $i+1$ tiene al dígito 2, éste tiene asociado el peso 2^{i+1} y representa el valor $2(2^{i+1}) = 2^{i+1} + 2^{i+1} = 2(2^i) + 2^{i+1}$. Al ver esta igualdad, podemos darnos cuenta que el dígito 0 se debe eliminar sustituyéndolo por un 2 y cambiando el dígito 2 en la posición $i+1$ por un dígito 1.

Adaptando esta idea, recursivamente, obtenemos la siguiente definición para nuestras listas de acceso aleatorio:

```
zero :: HRAL (Fork bush) a -> HRAL bush a
zero Nil = Nil
zero (One b ds) = Two b (zero ds)
zero (Two (Fork b1 b2) ds) = Two b1 (One b2 ds)
```

Con la ayuda de esta función, la separación en cabeza y cola es directa:

```
uncons :: RAL a -> Maybe (a, RAL a)
uncons Nil = Nothing
uncons (One (Leaf a) ds) = Just (a, zero ds)
uncons (Two (Fork (Leaf a) b) ts) = Just (a, One b ts)
```

Pasamos ahora a discutir las funciones de acceso. La idea para acceder a un elemento en una lista de acceso aleatorio, es la misma: Primero buscamos el árbol donde se encuentra el elemento y posteriormente buscamos el elemento dentro de dicho árbol. La función `accessHRAL` se encarga de buscar el árbol correspondiente, si la lista empieza con `One` y buscamos el elemento 0, devolvemos el árbol asociado al primer dígito. En otro caso, buscamos el árbol en la posición $\lfloor \frac{i-1}{2} \rfloor$ en el resto de la lista, en dicha posición, necesariamente hay un árbol de la forma `Fork x y`. De acuerdo a la paridad de i se devuelve uno de los árboles `x` o `y`

La idea es equivalente cuando la lista empieza con el dígito `Two`. Los elementos que pueden ser accedidos de inmediato son los elementos 0 y 1; los árboles asociados al dígito `Two`. En otro caso, buscamos en el resto de la lista, buscando el elemento $\lfloor \frac{i-2}{2} \rfloor$. El código correspondiente es:

```
accessHRAL :: Integer -> HRAL bush a -> bush a
accessHRAL 0 (One t _) = t
accessHRAL i (One _ ts) = let
                            Fork x y = accessHRAL ((i-1) `div` 2) ts
                          in
                            if ((i-1) `mod` 2) == 0 then x else y
accessHRAL 0 (Two (Fork t1 _) _) = t1
accessHRAL 1 (Two (Fork _ t2) _) = t2
accessHRAL i (Two t ts) = let
                            Fork x y = accessHRAL ((i-2) `div` 2) ts
                          in
                            if ((i-2) `mod` 2) == 0 then x else y
```

Para obtener la función de acceso para el tipo `RAL`, simplemente se invoca a la función `accessHRAL` con el constructor `Leaf`:

```
access :: Integer -> RAL a -> a
access i ral = let Leaf a = accessHRAL i ral in a
```

La operación de actualización realizarse de manera análoga a la versión de las listas de acceso aleatorio basada en el sistema binario común, como se vio en la página 38. Sin embargo, queremos presentar una versión más elegante y eficiente.

En la versión anterior, debíamos hacer una llamada a la función de búsqueda para poder realizar la operación de actualización. Esto debía hacerse para poder construir el nuevo árbol que sustituiría a un árbol en la lista original. El nuevo árbol solo difería en una hoja. ¿Podemos evitar esa llamada a la función de búsqueda? La respuesta es sí, ¿cómo? construimos una función *al vuelo* que recibirá un árbol y devolverá la versión modificada. Naturalmente el caso base de esta función será la sustitución de una hoja. El caso donde la función deberá cambiar es cuando se hace una llamada recursiva sobre la cola de la lista. En este caso, se actualizará el árbol izquierdo o derecho, según la paridad del índice que se reciba.

```
fupdateHRAL :: (bush a -> bush a) -> Integer -> HRAL bush a -> HRAL
     bush a
fupdateHRAL f 0 (One t xs ) = One (f t) xs
fupdateHRAL f i (One t ts) = incr t (fupdateHRAL f (i-1) (zero ts))
fupdateHRAL f 0 (Two (Fork b1 b2) ts) = Two (Fork (f b1) b2) ts
fupdateHRAL f 1 (Two (Fork b1 b2) ts) = Two (Fork b1 (f b2)) ts
fupdateHRAL f i (Two t ts) =
           let
             f' = \(Fork x y) -> if ((i-2) `mod` 2) == 0 then Fork
                 (f x) y else Fork x (f y)
           in
             Two t (fupdateHRAL f' ((i-2) `div` 2) ts)
```

La función de actualización para listas de acceso aleatorio se implementa fácilmente con una llamada a la función `fupdateHRAL` con la función que transforma cualquier hoja en la hoja cuya etiqueta es el elemento a actualizar.

```
update :: Integer -> a -> RAL a -> RAL a
update i a ral = fupdateHRAL (\(Leaf x) -> (Leaf a)) i ral
```

Por último, analicemos como convertir una lista de acceso aleatorio a una lista común. El proceso inverso será, como en todas las versiones anteriores, una instancia del operador `foldr`.

En nuestra versión anterior, mostramos un proceso intuitivo pero ineficiente para convertir una LAA con tipos de datos anidados a una lista común. Regresemos a la implementación que usamos con las LAA sin usar tipos de datos anidados: aplanar árboles y concatenarlos. Adaptando esta idea a los tipos de datos anidados obtenemos la siguiente definición:

```
flatten (Leaf a) = [a]
flatten (Fork l r) = flatten l ++ flatten r
```

Sin embargo, esta función no está bien definida, puesto que, `Leaf a` y `Fork l r` no tienen el mismo tipo a pesar de que en conjunto representan árboles binarios perfectos. Además, como mencionamos antes, esta operación no es eficiente.

Este inconveniente se corrije implementando una función para cada tipo:

```
unleaf:: Leaf a -> [a]
unleaf (Leaf x) = [x]

unfork :: (bush a -> [a]) -> (Fork bush a -> [a])
unfork flatten (Fork l r) = flatten l ++ flatten r
```

La definición de `unfork` resulta más complicada de lo esperado debido al anidamiento de tipos. La idea detrás de su definición en un lenguaje más intuitivo es: Si sabemos como aplanar árboles de tipo `bush a` entonces podemos definir como aplanar árboles de tipo `Fork bush a`. De esta manera, la función polimórfica `unfork`^{n} `unleaf` se encarga de aplanar árboles de tipo `Fork`^{n} `Leaf a`.

Con esta misma idea podemos presentar la función `listify` que se sirve de `unfork` y aplana listas de acceso aleatorio.

```
listify : (bush a -> [a]) -> (RAL bush a -> [a])
listify _ Nil = []
listify flatten (One b ds) = flatten b ++ listify (unfork flatten)
    ds
listify flatten (Two b ds) = unfork flatten b ++ listify (unfork
    flatten) ds
```

El primer argumento de la función `listify` es un “aplanador” de árboles de tipo `bush a`, el cual se actualiza en la llamada recursiva mediante la función `unfork`, para obtener un “aplanador” de árboles de tipo `Fork bush a`.

Nuevamente, la función `toList` para listas de acceso aleatorio es un caso particular de `listify` que utiliza la función `unleaf` como aplanador de hojas. Así la implementación final es:

```
toList :: RAL a -> [a]
toList s = listify unleaf s
```

Las implementaciones de este capítulo, en particular, la implementación de `unfork` y `listify` dejan ver que una implementación con tipos de datos anidados, es más elaborada que una implementación convencional. Esto se debe a que un tipo de datos anidado, realmente está definiendo a una familia infinita de tipos, los cuales no son independientes, sino que, están entrelazados por su definición. Debido a esto las funciones que involucran a un tipo anidado, realmente son familias infinitas de funciones entrelazadas. Por ejemplo, la función `listify` para `HRAL bush a` utiliza en la llamada recursiva a la función `listify` para `HRAL (Fork bush) a`, la cual a su vez llama a la función `listify` para `HRAL (Fork (Fork bush)) a`, etc, etc. Esto implica en particular que las signaturas de esta clase de funciones sean complicadas puesto que requieren funciones auxiliares como el caso de `flatten` para la función `listify`. Dado que, cuando se usan tipos anidados, las signaturas son obligatorias es importante tratar de simplificarlas. Esto se puede hacer mediante el mecanismo de clases de HASKELL. Ejemplificamos la idea para el caso de la función `listify`, lo cual resulta en una versión más elegante de dicha operación.

Considérese la clase `Flatten` definida como sigue:

```
class Flatten bush where
      flatten :: bush a -> [a]
```

Se observa que esta es una clase de orden superior puesto que sus elementos son constructores de tipos y no tipos. Los constructores de la clase `Flatten` son aquellos constructores de árbol `bush` que tienen definida una función `flatten:: bush a ->[a]`, es decir, los constructores de árbol que soportan una función de aplanamiento.

Nuestros constructores de tipo `Leaf` y `Fork` se pueden instanciar a esta clase como sigue:

```
instance Flatten Leaf where
      flatten (Leaf a) = [a]

instance (Flatten bush) => Flatten (Fork bush) where
      flatten (Fork l r) = flatten l ++ flatten r
```

Se observa que el constructor `Fork bush` solo puede pertenecer a la clase `flatten` si `bush` es miembro de la misma clase. Usando esta clase la función `listify` se simplifica como sigue:

```
listify :: (Flatten bush) => HRAL bush a -> [a]
listify Nil = []
listify (One b ds) = flatten b ++ listify ds
listify (Two b ds) = flatten b ++ listify ds
```

Podemos notar que el lado derecho es idéntico en ambos casos recursivos. Esto es porque la función `flatten` está sobrecargada debido al mecanismo de clases. En el primer caso su tipo es `bush a ->[a]` mientras que en el segundo caso tiene tipo `Fork bush a ->[a]`. También es importante notar que el trato uniforme de ambos casos se debe a que elegimos representar los dos árboles asociados al dígito `Two` con un único árbol. Ahora bien, la función `toList` se convierte en `listify`.

```
toList :: RAL a -> [a]
toList s = listify s
```

Con esto terminamos nuestra implementación de listas de acceso aleatorio utilizando tipos de datos anidados, la cual, además de tener todas las ventajas mencionadas en el capítulo anterior, tiene la propiedad de garantizar estáticamente, es decir, mediante el tipado, que las listas manipuladas sean listas de acceso aleatorio válidas.

Conclusiones

En este trabajo hemos presentado a la estructura de datos conocida como lista de acceso aleatorio, la cual, ofrece al programador funcional operaciones eficientes de búsqueda y actualización de elementos. Nuestras implementaciones son ejemplos de representaciones numéricas basadas en sistemas numéricos binarios. Es decir, una estructura de tamaño n se obtiene a partir de la representación binaria del número n. En total desarrollamos 5 implementaciones que incluyen funciones de acceso eficientes (de $O(\log n)$) resumidas a continuación:

- *LAA basadas en el sistema binario estándar.* Esta implementación causa ineficiencias en las operaciones de listas comunes.
- *LAA basadas en sistema binario sin cero.* Esta implementación mejora la versión anterior, recuperando la eficiencia de la función `head`.
- *LAA basadas en el sistema binario sesgado.* Se recupera la eficiencia de las funciones de listas comunes. Esta implementación resulta ser la más eficiente de todas.
- *LAA con tipos anidados de primer orden, basadas en el sistema binario estándar.* Esta implementación ofrece una funcionalidad igual a nuestra primera implementación pero capturando estáticamente la noción de LAA válida.
- *LAA con tipos anidados de orden superior, basadas en el sistema binario sin cero.* Esta implementación ofrece una funcionalidad igual a nuestra segunda implementación pero capturando estáticamente la noción de LAA válida mediante el uso de constructores de tipo.

Debido a que en todos los casos, las operaciones de acceso resultan eficientes, cualquiera de estas implementaciones puede servir como una implementación funcional de arreglos. Más aún, en nuestro caso no se requiere a priori una cota para el tamaño inicial del arreglo, puesto que este puede crecer arbitrariamente. Es importante observar que a pesar de que nuestras LAA resultan ser eficientes para implementar arreglos, sus tiempos de ejecución son inferiores a los correspondientes para el caso de arreglos imperativos, aquellos son $O(\log n)$ y éstos son $O(1)$. Para una discusión más profunda sobre el concepto de arreglo funcional y sus implementaciones veáse [1]

Hemos dicho que las operaciones definidas sobre las listas de acceso aleatorio, a excepción de las funciones de acceso, son análogas a funciones sobre el sistema numérico, base de la definición de dichas listas. Sin embargo, surge la duda ¿qué ocurre con la función suma? Esta operación, en las listas comunes, corresponde a la función de concatenación y se define de la siguiente manera:

```
append :: List a -> List a -> List a
append Nil l = l
append (Cons x xs) l = Cons x (append xs l)
```

Que claramente es análoga a la función suma en el sistema unario, definida de la siguiente manera:

```
add :: Nat -> Nat -> Nat
add Zero m = m
add (Succ n) m = Succ (add n m)
```

En este sistema numérico, la función `append` es efectivamente análoga a la función `add`. Sin embargo, cuando cambiamos de sistema numérico, el algoritmo usual para sumar consiste en sumar dígito a dígito, llevando un acarreo tal como lo hacemos en el sistema decimal usual.

Siguiendo esta idea, podríamos implementar la función `append` en las listas de acceso aleatorio, pero al sumar dos dígitos no nulos, nos veríamos obligados a combinar los árboles que éstos tienen asociados. Resultando esto, en una mezcla de los elementos, situación indeseable puesto que el resultado dela función `append` debe ser una lista que contiene en primera instancia a los elementos de la primera LAA seguidos de los de la segunda.

Esta idea funciona en otras representaciones numéricas, como *montículos binomiales*, donde el orden en el que se guarden los elementos no importa, a excepción de preservar que el elemento mínimo o máximo del árbol esté en la raíz.

Como una posible línea de trabajo futuro, mencionamos la implementación de listas de acceso aleatorio con tipos anidados basadas en el sistema binario sesgado. Recordemos que para esta implementación, necesitamos árboles con información en todos los nodos, los cuales pueden definirse de la siguiente manera:

```
data Perfect bush a = Zero (bush a)
                    | Succ (Perfect (Fork bush) a)

data Fork bush a = Fork (bush a) a (bush a)
```

Obsérvese que lo único que se hizo con respecto a la definición anterior de `Perfect` y `Fork` fue modificar el constructor de árboles `Fork` para que incluya información en el nodo raíz, con lo cual se consiguen árboles con información en todos los nodos. El principal obstáculo para esta implementación radica en el hecho de que el sistema empleado es disperso, por lo que no se pueden hacer explícitos los dígitos nulos y esta característica resulta muy difícil de capturar con la clase de tipos anidados que hemos utilizado hasta ahora.

Apéndice

LAA con sistema numérico estándar

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}
module RALStandard where
import Prelude hiding (lookup, replicate)
import RandomAccessList


data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)

data Digit a = Zero
            | One (Tree a)


data RAL a = Nil
           1| Cons (Digit a) (RAL a)


instance Show a => Show (Tree a) where
    show (Leaf x) = "Leaf " ++ (show x)
    show (Node _ t1 t2) = "Node (" ++ (show t1) ++ ", " ++ (show t2
        ) ++")"
instance Show a => Show (Digit a) where
    show (Zero) = "Zero"
    show (One t) = "One (" ++ (show t) ++ ")"

instance Show a => Show (RAL a) where
    show l = show (tolst l)
        where
        tolst Nil = []
        tolst (Cons d l) = d : (tolst l)


link :: Tree a -> Tree a -> Tree a
link (Leaf x1) (Leaf x2) = Node 1 (Leaf x1) (Leaf x2)
link t1@(Node n _ _) t2@(Node m _ _)
    | n == m = Node (n+1) t1 t2
    | otherwise = error "n != m"
```

```
size :: Tree a -> Int
size (Leaf _) = 1
size (Node n _ _) = 2^n

flat :: Tree a -> [a]
flat (Leaf x) = [x]
flat (Node _ t1 t2) = flat t1 ++ flat t2

consTree :: Tree a -> RAL a -> RAL a
consTree t Nil = Cons (One t) Nil
consTree t (Cons Zero ts) = Cons (One t) ts
consTree t1 (Cons (One t2) ts) = Cons Zero (consTree (link t1 t2)
    ts)

unconsTree :: RAL a -> (Tree a, RAL a)
unconsTree (Cons (One t) Nil) = (t, Nil)
unconsTree (Cons (One t) ts ) = (t, Cons Zero ts)
unconsTree (Cons   Zero  ts ) =
    let (Node _ t1 t2, ts') = unconsTree ts
    in (t1, Cons (One t2) ts')

lookupTree :: Tree a -> Int -> a
lookupTree (Leaf x) 0 = x
lookupTree t@(Node _ t1 t2) i
    | i < (size t) `div` 2 = lookupTree t1 i
    | otherwise = lookupTree t2 (i - ((size t) `div` 2))


updateTree :: Tree a -> Int -> a -> Tree a
updateTree (Leaf x) 0 y = Leaf y
updateTree t@(Node n t1 t2) i y
    | i < (size t) `div` 2 = Node n (updateTree t1 i y) t2
    | otherwise = Node n t1 (updateTree t2 (i - ((size t) `div` 2))
         y)


uncons :: RAL a -> (a, RAL a)
uncons ts = let (Leaf x, ts') = unconsTree ts in (x, ts')

instance RandomAccessList.RALClass RAL a where

    --cons :: a -> RAL a -> RAL a
    cons x ts = consTree (Leaf x) ts

    head = fst.uncons
    tail = snd.uncons

    -- lookup :: RAL a -> Int -> a
    lookup (Cons  Zero   ts) i = lookup ts i
    lookup (Cons (One t) ts) i | i < size t = lookupTree t i
                               | otherwise  = lookup ts (i - size t
                                   )

    -- update :: RAL a -> Int -> a -> RAL a
    update (Cons Zero  ts) i y = update ts i y
    update (Cons (One t) ts) i y | i < size t = Cons (One (
```

```
        updateTree t i y)) ts
                                | otherwise  = Cons (One t) (
                                    update ts (i - size t) y)


    --fromList :: [a] -> RAL a
    fromList = foldr cons Nil

    --toList :: RAL a -> [a]
    toList   Nil = []
    toList (Cons Zero ts) = toList ts
    toList (Cons (One t) ts) = (flat t) ++ toList ts


replicate :: Int -> Tree a -> RAL a
replicate 0 t = Nil
replicate n t@(Node r t1 t2) =
    if n `mod` 2 == 1 then
        Cons (One t) (replicate (n `div` 2) (Node (r+1) t t))
    else
        Cons  Zero  (replicate (n `div` 2) (Node (r+1) t t))


modDiv2 :: [Tree a] -> (Digit a, [Tree a])
modDiv2 [] = (Zero, [])
modDiv2 (t:ts) = case modDiv2 ts of
                                    (Zero, q) -> (One t, q)
                                    (One t', q) -> (Zero, (link t t
                                        '):q)

fromListV2 :: [a] -> RAL a
fromListV2 l = fromListAux (map (\x->Leaf x) l)

fromListAux :: [Tree a] -> RAL a
fromListAux [] = Nil
fromListAux l = Cons d (fromListAux ts)
                                where (d,ts) = modDiv2 l
```

LAA. con sistema numérico 1,2

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}
module RALOneTwo where
import Prelude hiding (lookup)
import RandomAccessList

data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)

data Digit a = One (Tree a)
             | Two (Tree a) (Tree a)

data RAL a = Nil
           | Cons (Digit a) (RAL a)
```

```
instance Show a => Show (Tree a) where
  show (Leaf x) = "Leaf " ++ (show x)
  show (Node _ t1 t2) = "Node (" ++ (show t1) ++ ", " ++ (show t2)
      ++")"
instance Show a => Show (Digit a) where
  show (One t) = "One (" ++ (show t) ++ ")"
  show (Two t1 t2) = "Two <" ++ (show t1) ++ ", " ++ (show t2)
      ++">"

instance Show a => Show (RAL a) where
  show l = show (tolst l)
    where
    tolst Nil = []
    tolst (Cons d l) = d : (tolst l)


link :: Tree a -> Tree a -> Tree a
link (Leaf x1) (Leaf x2) = Node 1 (Leaf x1) (Leaf x2)
link t1@(Node n _ _) t2@(Node m _ _) | n == m = Node (n+1) t1 t2
                             | otherwise = error "n != m"

size :: Tree a -> Int
size (Leaf _) = 1
size (Node n _ _) = 2^n

flat :: Tree a -> [a]
flat (Leaf x) = [x]
flat (Node _ t1 t2) = flat t1 ++ flat t2

consTree :: Tree a -> RAL a -> RAL a
consTree t Nil = Cons (One t) Nil
consTree t1 (Cons (One t2) ts) = Cons (Two t1 t2) ts
consTree t1 (Cons (Two t2 t3) ts) = Cons (One t1) (consTree (link
    t2 t3) ts)

unconsTree :: RAL a -> (Tree a, RAL a)
unconsTree (Cons (One t) Nil) = (t, Nil)
unconsTree (Cons (Two t1 t2) Nil) = (t1, Cons (One t2) Nil)
unconsTree (Cons (One t) ts) = let (Node _ t1 t2, ts') = unconsTree
     ts in (t, Cons (Two t1 t2) ts')
unconsTree (Cons (Two t1 t2) ts) = (t1, Cons (One t2) ts)


lookupTree :: Tree a -> Int -> a
lookupTree (Leaf x) 0 = x
lookupTree t@(Node _ t1 t2) i | i < (size t) `div` 2 = lookupTree
    t1 i
                              | otherwise = lookupTree t2 (i - ((
                                  size t) `div` 2))


updateTree :: Tree a -> Int -> a -> Tree a
updateTree (Leaf x) 0 y = Leaf y
updateTree t@(Node n t1 t2) i y
          | i < (size t) `div` 2 = Node n (updateTree t1 i y) t2
          | otherwise = Node n t1 (updateTree t2 (i - ((size t) `
```

```
            div‘ 2)) y)


uncons :: RAL a -> (a, RAL a)
uncons ts = let (Leaf x, ts’) = unconsTree ts in (x, ts’)


instance RandomAccessList.RALClass RAL a where

  --cons :: a -> RAL a -> RAL a
  cons x ts = consTree (Leaf x) ts

  head = fst.uncons
  tail = snd.uncons

  -- lookup :: RAL a -> Int -> a
  lookup (Cons (One t) ts) i | i < size t = lookupTree t i
                             | otherwise = lookup ts (i - size t)
  lookup (Cons (Two t1 t2) ts) i | i < size t1 = lookupTree t1 i
                                 | i < (size t2)*2 = lookupTree t2
                                     (i-(size t1))
                                 | otherwise = lookup ts (i - (size
                                     t1)*2)

  -- update :: RAL a -> Int -> a -> RAL a
  update (Cons (One t) ts) i y | i < size t = Cons (One (updateTree
       t i y)) ts
                               | otherwise =  Cons (One t) (update
                                   ts (i - size t) y)
  update (Cons (Two t1 t2) ts) i y | i < size t1 =
                                     Cons (Two (updateTree t1 i y)
                                          t2) ts
                                   | i < (size t2)*2 =
                                     Cons (Two t1 (updateTree t2
                                         (i - (size t1)) y)) ts
                                   | otherwise =
                                     Cons (Two t1 t2) (update ts (
                                         i - (size t1)*2) y)

  --fromList :: [a] -> RAL a
  fromList = foldr cons Nil

  -- toList :: RAL a -> [a]
  toList Nil = []
  toList (Cons (One t) ts) = flat t ++ toList ts
  toList (Cons (Two t1 t2) ts) = flat t1 ++ flat t2 ++ toList ts
```

LAA con sistema numérico sesgado

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}
module RALSkew where
import Prelude hiding (lookup)
import Data.List hiding (lookup)
import RandomAccessList
```

```
data Tree a = Leaf a
            | Node (Tree a) a (Tree a)


data RAL a = Nil
                                          | Cons (Int, Tree a) (RAL
                                            a)

flat (Leaf a) = [a]
flat (Node t1 x t2) = x : (flat t1 ++ flat t2)

lookupTree :: Int -> Tree a -> Int -> a
lookupTree 1 (Leaf x) 0 = x
lookupTree _ (Node t1 x t2) 0 = x
lookupTree w (Node t1 x t2) i
             | i <= w = lookupTree (w `div` 2) t1 (i-1)
             | otherwise = lookupTree (w `div` 2) t2 (i - 1 - (w `
                 div` 2))

updateTree :: Int -> Tree a -> Int -> a -> Tree a
updateTree 1 (Leaf x) 0 y  = Leaf y
updateTree _ (Node t1 x t2) 0 y = Node t1 y t2
updateTree w (Node t1 x t2) i y
             | i <= w = updateTree (w `div` 2) t1 (i-1) y
             | otherwise = updateTree (w `div` 2) t2 (i - 1 - (w `
                 div` 2)) y



instance RandomAccessList.RALClass RAL a where

        cons x ts@(Cons (w1,t1) (Cons (w2,t2) ts'))
                                                                    |

                                                                       w1

                                                                       ==

                                                                       w2

                                                                       =

                                                                       Cons

                                                                       (1+
                                                                       w1
                                                                       +
                                                                       w2
                                                                       ,

                                                                       (
                                                                       Node

                                                                       t1

                                                                       x
```

```
t2
)
)

ts
'

|

otherwise

=

Cons

(1,

Leaf

x
)

ts
```

```
cons x l = Cons (1, Leaf x) l

head (Cons (1, Leaf x) _) = x
head (Cons (w, Node _ x _) _) = x

tail (Cons (1, Leaf _) ts) = ts
tail (Cons (w, Node t1 x t2) ts) = Cons (w `div` 2, t1) (
    Cons (w `div` 2, t2) ts)


lookup (Cons (w, t) ts) i
                                | i < w = lookupTree w t i
                                | otherwise = lookup ts (i-
                                  w)

update (Cons (w, t) ts) i y
                                | i < w = Cons (w,
                                  updateTree w t i y) ts
                                | otherwise = Cons (w, t) (
                                  update ts (i-w) y)

fromList = foldr cons Nil

toList ts = concat (map (\(w,t)-> flat t) (unfoldr tolst ts
    ))
        where
        tolst Nil = Nothing
        tolst (Cons x xs) = Just (x, xs)
```

LAA con TDA de primer orden

```
{-# LANGUAGE TypeSynonymInstances #-}
```

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}
module RALNestedFO where
import Prelude hiding (lookup)
import RandomAccessList

data RAL a = Nil
           | Zero (RAL (a,a))
           | One a (RAL (a,a))
           deriving Show

len :: RAL a -> Integer
len Nil = 0
len (Zero l) = 2*(len l)
len (One _ xs) = 1 + 2*(len xs)

uncons :: RAL a -> (a, RAL a)
uncons (One x Nil) = (x, Nil)
uncons (One x ps) = (x, Zero ps)
uncons (Zero ps) = (x, One y ps') where ((x,y), ps') = uncons ps

instance RandomAccessList.RALClass RAL a where

  --cons :: a -> RAL a -> RAL a
  cons x Nil = One x Nil
  cons x (Zero ps) = One x ps
  cons x (One  y ps) = Zero (cons (x,y) ps)


  head xs = x where (x, _) = uncons xs
  tail xs = xs' where (x,xs') = uncons xs

  --lookup :: RAL a -> Integer -> a
  lookup (One x _)  0 = x
  lookup (One _ xs) i = lookup (Zero xs) (i-1)
  lookup (Zero xs)  i = let (x,y) = lookup xs (i `div` 2)
                        in if (i `mod` 2) == 0 then x else y

  --update :: RAL a -> Integer -> a -> RAL a
  update (One x xs) 0 e = One e xs
  update (One x xs) i e = cons x (update (Zero xs) (i-1) e)
  update (Zero xs)  i e = let
                         (x,y) = lookup xs (i `div` 2)
                         p = if (i `mod` 2) == 0 then (e,y) else
                             (x,e)
                        in
                         Zero (update xs (i-1) p)

  --fromList :: [a] -> RAL a
  fromList = foldr cons Nil

  --toList :: RAL a -> [a]
  toList Nil = []
  toList l = let (x, xs) = uncons l in x:(toList xs)
```

```
superUpdate :: Integer -> a -> RAL a -> RAL a
superUpdate i e xs = fupdate (\x -> e) i xs

fupdate :: (a -> a) -> Integer -> RAL a -> RAL a
fupdate f 0 (One x xs) = One (f x) xs
fupdate f i (One x xs) = cons x (fupdate f (i-1) (Zero xs))
fupdate f i (Zero xs) = let
                          f' = \(x,y) -> if (i `mod` 2) == 0 then (
                              f x, y) else (x, f y)
                        in
                          Zero (fupdate f' (i `div` 2) xs)
```

LAA con TDA de orden superior

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}
module RALNestedHighOrder where
import Prelude hiding (lookup)
import RandomAccessList

data Leaf a = Leaf a
data Fork bush a = Fork (bush a) (bush a)

data RAL bush a =
       Nil
     | One (bush a) (RAL (Fork bush) a)
     | Two (Fork bush a) (RAL (Fork bush) a)

type IxSequence = RAL Leaf

class Flatten bush where
      flatten :: bush a -> [a]

instance (Show a) => Show (Leaf a) where
   show (Leaf i) = "Leaf " ++ (show i)

instance (Show (bush a), Show a) => Show (Fork bush a) where
   show (Fork f1 f2) = "Fork [" ++ (show f1) ++ "," ++ show f2 ++
       "]"

instance (Show (bush a), Show a) => Show (RAL bush a) where
   show Nil = "nil"
   show (One t l) = "One <" ++ (show t) ++ "> (" ++ (show l) ++ ")"
   show (Two t l) = "Two <" ++ (show t) ++ "> (" ++ (show l) ++ ")"

instance Flatten Leaf where
      flatten (Leaf a) = [a]

instance (Flatten bush) => Flatten (Fork bush) where
      flatten (Fork l r) = flatten l ++ flatten r

incr :: bush a -> RAL bush a -> RAL bush a
incr b Nil = One b Nil
incr b1 (One b2 ds) = Two (Fork b1 b2) ds
incr b1 (Two b2 ds) = One b1 (incr b2 ds)
```

```
zero :: RAL (Fork bush) a -> RAL bush a
zero Nil = Nil
zero (One b ds) = Two b (zero ds)
zero (Two (Fork b1 b2) ds) = Two b1 (One b2 ds)

front :: IxSequence a -> Maybe (a, IxSequence a)
front Nil = Nothing
front (One (Leaf a) ds) = Just (a, zero ds)
front (Two (Fork (Leaf a) b) ts) = Just (a, One b ts)

accessRAL :: Int -> RAL bush a -> bush a
accessRAL 0 (One t _) = t
accessRAL i (One _ ts) = let
                           Fork x y = accessRAL ((i-1) `div` 2) ts
                         in
                           if ((i-1) `mod` 2) == 0 then x else y
accessRAL 0 (Two (Fork t1 _) _) = t1
accessRAL 1 (Two (Fork _ t2) _) = t2
accessRAL i (Two t ts) = let
                           Fork x y = accessRAL ((i-2) `div` 2) ts
                         in
                           if ((i-2) `mod` 2) == 0 then x else y

fupdateRAL :: (bush a -> bush a) -> Int -> RAL bush a -> RAL bush a
fupdateRAL f 0 (One a xs ) = One (f a) xs
fupdateRAL f i (One t ts) = incr t (fupdateRAL f (i-1) (zero ts))
fupdateRAL f 0 (Two (Fork b1 b2) ts) = Two (Fork (f b1) b2) ts
fupdateRAL f 1 (Two (Fork b1 b2) ts) = Two (Fork b1 (f b2)) ts
fupdateRAL f i (Two t ts) =
           let
             f' = \(Fork x y) -> if ((i-2) `mod` 2) == 0 then Fork
                 (f x) y else Fork x (f y)
           in
             Two t (fupdateRAL f' ((i-2) `div` 2) ts)


unleaf :: Leaf a -> [a]
unleaf (Leaf a) = [a]

unfork :: (bush a -> [a]) -> (Fork bush a -> [a])
unfork flatten (Fork l r) = flatten l ++ flatten r

listify :: (Flatten bush) => RAL bush a -> [a]
listify Nil = []
listify (One b ds) = flatten b ++ listify ds
listify (Two b ds) = flatten b ++ listify ds


instance RandomAccessList.RALClass IxSequence a where

  -- cons :: a -> IxSequence a -> IxSequence a
  cons a s = incr (Leaf a) s

  head l = case front l of
              Nothing -> error "Empty list"
              Just (x,_) -> x
```

```
tail l = case front l of
            Nothing -> error "Empty list"
            Just (_,xs) -> xs

-- lookup :: IxSequence a -> Int -> a
lookup ral i = let Leaf a = accessRAL i ral in a

-- update :: IxSequence a -> Int -> a -> IxSequence a
update ral i a = fupdateRAL (\(Leaf x) -> (Leaf a)) i ral

-- fromList :: [a] -> IxSequence a
fromList = foldr cons Nil

-- toList :: IxSequence a -> [a]
toList s = listify s
```

Bibliografía

[1] Grue Klaus E.: *Arrays in Pure Functional Programming Languages*, University of Copenhagen 1989

[2] Hinze Ralph: *Numerical representations as Higher Order Nested Datatypes*, Institut für Informatik III. Universität Bonn 1998.

[3] Hutton Graham: *Programming in Haskell* Cambridge University Press, 2007.

[4] Ivanovic Mirjana, Kuncak Viktor: *Numerical representations as Purely Functional Data Structures: a New approach*, Faculty of Science and Mathematics, University of Novi Sad 2001.

[5] Myers Eugene W.: *An applicative random access stack*, Information Processing Letters, 17(5):241–248, 1983.

[6] Okasaki Chris: *Purely functional data structures* Cambridge University Press, 1998.

[7] Okasaki Chris: *Purely functional random-access lists*, En *Conference on Functional Programming Languages and Computer Architecture*, Junio 1995.