Universidad Nacional Autónoma de México

Facultad de Ciencias

Programación Origami: Cómo Doblar un Árbol

T E S I S

que para obtener el título de:

Licenciado en Ciencias de la Computación

PRESENTA:

Javier Enríquez Mendoza

TUTORA

Dra. Lourdes Del Carmen González Huesca

CD. MX. 2020

Hoja de Datos del Jurado

1. Datos del Alumno
   Enríquez
   Mendoza
   Javier
   477 645 4557
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la Computación
   415000073

---

2. Datos de la Tutora
   Dra.
   Lourdes del Carmen
   González
   Huesca

---

3. Datos del Sinodal 1
   Dr.
   Favio Ezequiel
   Miranda
   Perea

---

4. Datos del Sinodal 2
   Dra.
   Adriana
   Ramírez
   Vigueras

---

5. Datos del Sinodal 3
   M. en C.
   Pilar Selene
   Linares
   Arévalo

---

6. Datos del Sinodal 4
   M. en I.
   Karla
   Ramírez
   Pulido

---

7. Datos del trabajo escrito
   Programación Origami: Cómo Doblar un Árbol
   100p.
   2020

Agradecimientos

A Iliana y Javier. Este logro es tan suyo como lo es mío, por siempre apoyarme y procurarme. Por enseñarme a soñar al mismo tiempo que mantenían mis pies en la tierra, por siempre esperar grandes cosas de mi y creer que era capaz de lograrlo. Por todos los sacrificios, por ponerme a mi antes que a ustedes mismos en tantas ocasiones y aún así no esperar nada a cambio. Por soportarme en mis peores momentos y estar a mi lado en mis logros. Por todo el amor y la paciencia. Jamás voy a terminar de agradecerles todo lo que hacen por mi y espero hacerlos sentir tan orgullosos de mi como yo me siento de ustedes.

A Fernanda, por crecer conmigo y ser mi ejemplo y mi guía, por ser (a veces) la mejor hermana, quererme, consentirme y apoyarme siempre.

A Lourdes, mi asesora, por todos los consejos, la paciencia, las enseñanzas y por ayudarme a vivir tantas experiencias increíbles, por todas esas reuniones y sobre todo por confiar en mi, por escucharme y entenderme, jamás hubiera podido lograr nada de esto de no ser por ti.

A Valeria, por tantos años de amistad, por permitirme abrirme completamente contigo y ser auténticamente yo, por ayudarme a crecer, por haber estado ahí en cada paso, por todas las risas y todos los momentos que hemos compartido y que no podría haber vivido con nadie más, por todas esas tardes en la biblioteca, por apoyarme tanto y por todo el cariño que muy a nuestra manera me has dado. Sobre todo, gracias por no prestarme ese lápiz.

A Mau, por ser de los mejores amigos que alguien puede tener, por ayudarme tanto en tantos aspectos, por todas esas platicas interminables en las que me permitías desahogarme contigo, esos viajes en metro, esos tacos, las clases y horas de estudio que si no hubieran sido contigo definitivamente no hubieran sido igual, por compartir tantas experiencias conmigo, por todo lo que me enseñaste y por siempre estar ahí conmigo dentro y fuera de la facultad.

A Manuel, por enseñarme todas esas experiencias que me daban miedo

y hoy disfruto tanto que sin ti probablemente jamás hubiera conocido, por todas esas platicas y chismes, por todo el apoyo y siempre guiarme y ayudarme. Pero sobre todo gracias por creer en mi en el momento en el que ni siquiera yo lo hacia y por empujarme hacia el camino correcto y siempre decirme justo lo que necesito escuchar.

A Rodro, por esa amistad tan increíble, por compartir tanto conmigo, por enseñarme lugares tan maravillosos para comer, por todas esas platicas y todos los momentos que vivimos juntos y sobre todo por toda la comida.

A Iker, por hacer mucho mas amenas todas las clases que compartimos y todos los momentos que pasamos en la facultad, por todas las risas y momentos divertidos, por todo el apoyo y también por ayudarme tanto a lo largo de la carrera a entender y ver las cosas de otra forma.

A Diego, por enseñarme que era un árbol cartesiano, por ese gran saludo y por compartir tantos momentos increíbles conmigo.

A todos los amigos me acompañaron en estos años, Andrea, Paoly, Caro, Gabriel, Karla, Nastia, Oscar, Cuadru, Ady, Marianas, Ricardo, Tabo.

A todos mis profesores, por haberme enseñado tanto a lo largo de la carrera, especialmente a Karla, Selene y Favio que a parte de ser los mejores profesores que tuve en la carrera y mostrarme esa parte de la Computación que tanto me gusta, confiaron en mi para ser su ayudante.

A cada uno de mis alumnos de los cuales he aprendido mucho más de lo yo haya podido enseñarles. Este trabajo está en gran parte inspirado en ustedes, en todas esas clases que tuvimos juntos, en sus dudas de las cuales nunca deje de aprender. Gracias por enseñarme a enseñar. Especialmente gracias a la generación 2018 con la cual he compartido tantos momentos y he conocido a personas maravillosas.

Índice general

Motivación y estructura del trabajo

Origami (折り紙) es el arte japonés de crear elegantes diseños usando dobleces en cualquier tipo de papel, de *ori* (doblez) y *kami* (papel) [1]. Con base en este arte se crea un estilo de programación a partir de funciones de plegado (`fold`) definidas sobre estructuras de datos recursivas.

El objetivo principal de este trabajo es definir y estudiar las funciones de plegado sobre estructuras arbóreas, las cuales son ampliamente utilizadas en Ciencias de la Computación como una estructura para almacenar información, así como determinar cuándo una función puede reescribirse usando la técnica de programación origami, es decir, definirla como una instancia de los operadores `fold`.

El estudio la programación origami es relevante, pues el trabajo ya hecho sobre la estructura de datos lista ha demostrado ser de gran utilidad para mejorar la eficiencia de los programas, así como para simplificar el razonamiento ecuacional sobre éstos.

En este trabajo se estudian los operadores de plegado y sus propiedades, utilizando álgebras semánticas como mecanismo de abstracción de los constructores para tres tipos generales de árboles [2]:

1. Los árboles binarios tipo 1 (o con información sólo en las hojas), que se definen en el capítulo 3.
2. Los árboles binarios de tipo 2 (o con información en todos los nodos) estudiados en los capítulos 4, 5 y 6.
3. Los árboles n-arios que corresponden al capítulo 7.

El concepto de álgebra semántica se definirá más adelante en este trabajo, y se usará en todas las estructuras definidas. Fue elegido este método porque de esta forma resulta más sencillo abstraer las definiciones de funciones que usan el operador de plegado respecto a los constructores de las estructuras, en

comparación con la caza de patrones que es más común cuando se habla de programación origami.

El capítulo 8 del trabajo está dedicado a estructuras arbóreas más complejas y dado que estos ejemplos son casos particulares de las estructuras definidas en capítulos anteriores, sólo se define su estructura y la función `fold` correspondiente.

Finalmente, en el capítulo 9, se definen condiciones necesarias y suficientes para poder expresar una función en términos de los operadores de plegado.

Capítulo 1

Fundamentos

En este capítulo se definen algunas nociones necesarias para el desarrollo de este trabajo, así como los estilos de programación que se usaran para definir los operadores de plegado y una breve introducción a estructuras arbóreas con definiciones de conceptos básicos de éstas.

1.1. Programación declarativa

Los lenguajes de programación se pueden dividir en dos grandes categorías: la programación imperativa o procedimental y la programación declarativa. Estos estilos de programación, también llamados paradigmas, son opuestos.

En el estilo imperativo, los programas son una secuencia de instrucciones ejecutadas de forma lineal, es decir instrucción por instrucción. Un programa en este estilo establece cómo se resuelve el problema a atacar. Las características principales de este estilo son:

1. Manejo de datos usando estructuras de control como los ciclos `while`, `repeat`, `for`, etc.

2. Manejo de memoria a través de asignaciones de valores a variables.

Mientras que en el estilo declarativo los programas se ven como una sucesión de definiciones, siendo la recursión la principal estructura de control, no existen ciclos ni operaciones de asignación. Es decir, el programa define qué se debe calcular, siendo el cómo completamente irrelevante.

Algunas ventajas de la programación declarativa sobre la imperativa son[3]:

- Definiciones más cercanas al lenguaje matemático, haciéndolos más generales, cortos y legibles. Se puede decir que la programación declarativa es matemáticamente más elegante[1].
- Se favorece la verificación de los programas declarativos respecto a una especificación.
- Los programas son más fáciles de mantener, modificar y depurar.

Dentro de la programación declarativa tenemos otros estilos como son el funcional y el lógico. La programación origami es una técnica de programación del estilo funcional.

1.2. Programación Funcional

En la década de 1930 Alonzo Church define el Cálculo Lambda como un modelo teórico de cómputo basado en definiciones y aplicaciones de funciones. El Cálculo Lambda es equivalente a un máquina de Turing, como se demuestra en la Tesis de Church-Turing[4]. Este modelo junto con la Teoría de Categorías son considerados los fundamentos matemáticos sobre los cuales se explica y se desarrolla la programación funcional. Es gracias a esta base teórica que se tiene un razonamiento muy cercano al matemático sobre los programas desarrollados en lenguajes funcionales, este es el principal motivo por el cual se usa este estilo de programación en este trabajo.

Siguiendo la definición de Richard Bird en su libro *Thinking Functionally with Haskell*[5], la programación funcional:

- Es un método de programación que enfatiza el uso de funciones y aplicaciones más que el uso comandos y sus ejecuciones.
- Utiliza notación matemática que permite describir los problemas de forma clara y concisa.
- Tiene una base matemática simple que soporta razonamiento ecuacional sobre las propiedades del programa.

En un lenguaje de programación funcional, la orientación es hacia la evaluación dejando un poco de lado la ejecución, en otras palabras la programación funcional se centra en el resultado y no tanto en el cómo se obtuvo. Además, el concepto de composición de funciones es muy importante pues a partir de composición de funciones simples se generan nuevas y mas complejas[5].

Una propiedad de las funciones en los lenguajes funcionales es la *transparencia referencial*, es decir que se puede sustituir una expresión por otra de igual

[1] Según algunos autores.

valor sin afectar la semántica del programa. Así también, el valor de una función depende exclusivamente de los valores de sus argumentos. Como consecuencia de esto no es explícita la noción de estado o memoria en los lenguajes funcionales, lo cual podría verse como una desventaja. La noción de estado no es indispensable para la programación funcional ya que un programa solo describe la solución en general. El mecanismo principal de definición de funciones es la recursión. La inducción es la metodología para probar propiedades sobre éstos[6][7][8].

Algunos de los lenguajes de programación funcionales más utilizados en la actualidad son: Haskell, Lisp, Scheme, Racket, ML, entre muchos otros. Este trabajo se desarrolla en Haskell.

Haskell es un lenguaje de programación de propósito general, estáticamente tipado y puramente funcional con inferencia de tipos y evaluación perezosa. Está diseñado para ajustarse a la academia, la investigación y la industria, proporcionando un lenguaje de programación que incorpora la teoría de funciones (Cálculo Lambda) en un lenguaje elegante y poderoso para cualquier desarrollo. Debido a la pureza del lenguaje, Haskell no tiene efectos (colaterales[2]), es decir no hay efectos visibles además del valor de regreso de una función. Por ejemplo, no es posible realizar la manipulación directa de memoria[9][10].

Cualquier expresión en Haskell tiene un tipo asociado y éstos son inferidos automáticamente sin necesidad de declaraciones explícitas, obteniendo siempre el tipo más general de cualquier programa. Esta característica se logra gracias a que Haskell es un lenguaje fuertemente tipado[3] incluyendo un sistema de tipos, además de no realizar conversiones implícitas entre tipos. Esto asegura que un código bien tipado no tenga errores o *bugs*[9].

Otro aspecto muy importante de Haskell es su evaluación perezosa, esto quiere decir que las expresiones serán evaluadas hasta que sea estrictamente necesario. Uno de los principales usos de este tipo de evaluación es la implementación de estructuras de datos infinitas.

Por lo tanto los programas escritos en Haskell son robustos, concisos y correctos, es decir van a funcionar siempre de la forma en la que se esperaría que lo hicieran por su naturaleza declarativa[11]. La pureza del lenguaje es un factor determinate en la elección de Haskell para desarrollar este trabajo. Permite tener implementaciones sencillas y concisas,al mismo tiempo que facilita el razonamiento teórico. A partir de este punto se usará la sintaxis de Haskell, en las definiciones de código, la cual se irá explicando conforme se vaya utilizando.

[2]Del inglés side effects.
[3]Del correcto en español tipificado.

1.3. Funciones

Desde el punto de vista matemático las funciones son reglas de correspondencia que asocian cada elemento de un conjunto determinado A con un único elemento de un segundo conjunto B.

En la programación funcional, las funciones son entidades elementales, pues todo está modelado como una función[11].

Las funciones se pueden categorizar de diferentes formas dependiendo de las características de éstas. Para este trabajo vamos a estudiar con detalle dos categorías especificas:

- Funciones de Orden Superior.
- Funciones Anónimas.

Es importante aclarar que estas categorías no son ajenas ni totales, es decir, una función podría ser a la vez anónima y de orden superior o no pertenecer a ninguna de estas categorias.

Para un lenguaje funcional el nombre de una función no tienen ningún tratamiento especial, son tratadas como variables de tipo función, es decir, que para el lenguaje el nombre de una función es tratado de la misma forma que el nombre de un valor numérico[3].

1.3.1. Funciones de Orden Superior

Las funciones de Orden Superior son funciones que toman otras funciones como argumentos o regresan una función como resultado[12]. Para poder tener funciones de orden superior es necesario que las funciones sean elementos de primera clase. Una función es una entidad de primera clase de los lenguajes de programación funcional, ésto quiere decir que se pueden almacenar en estructuras de datos, utilizar como argumentos para otras funciones y regresar como resultado[3].

El uso principal de las funciones de orden superior es abstraer un comportamiento común. Existe una gran variedad funciones de orden superior ampliamente utilizadas en lenguajes de programación funcional. Quizá las más comunes sean `map`, `filter`, `foldr` y `foldl`. El programador puede también definir sus propias funciones de orden superior.

Para ilustrar mejor las Funciones de Orden Superior, tomamos como ejemplo la función `map` que aplica una función a todos los elementos de una lista. A continuación se muestra una especificación de `map` con la sintaxis de definición de funciones de Haskell

```
map : (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Esta Función es de Orden Superior pues como primer parámetro recibe otra función de tipo `a -> b`. Esta definición recursiva tiene como caso base aplicar `map` a una lista vacía y como caso recursivo aplicarlo a la lista con cabeza `x` y cola `xs`.

Si la función `map` se aplica parcialmente, es decir solo al recibir el primer argumento `f: a -> b` se obtendrá como resultado otra función `[a] -> [b]` espera una lista como argumento.

En muchas ocasiones para el paso de una función como parámetro o el regreso de una función como resultado se hace uso de funciones anónimas de las cuales hablamos en la siguiente sección.

1.3.2. Funciones Anónimas

Una función anónima es una función a la que no se le designa un nombre, es decir, no está relacionada a un identificador. También son conocidas como *lambdas* (λ).

Estas funciones se heredan de el Cálculo Lambda, en donde todos los cómputos se expresan usando abstracciones *lambda*, es decir, definiciones de funciones.

En algunas ocasiones es más conveniente usar funciones anónimas que definirlas con un nombre, por ejemplo cuando se está trabajando con Funciones de Orden Superior y en particular cuando no serán reutilizadas.

Siguiendo con el ejemplo de la sección anterior, se puede aplicar la función `map` a una función anónima.

```
map (\x -> x + 1) [1, 2, 3]
```

En donde `(\x -> x + 1)` es la función anónima en sintaxis de Haskell que incrementa en uno su argumento, en ella la primera `x` corresponde a la variable a ligar en el cuerpo de la función, que es `x + 1`. El resultado de la aplicación anterior es la lista `[2,3,4]`.

1.4. Programación origami

Un tipo de dato recursivo (inductivo) se define por medio de funciones constructoras, es decir la declaración de los elementos del tipo de dato se expresan

con reglas de construcción, las cuales se identifican por medio de un nombre exclusivo [13].

En la programación funcional, las funciones u operadores `fold` encapsulan el patrón de diseño con el que se definen las funciones recursivas sobre tipos de dato inductivos, manejando cada constructor como un caso de las funciones. Los casos base corresponden a los constructores que no dependen del mismo tipo y los casos recursivos a los constructores que sí dependen del mismo tipo de dato.

La programación origami hace uso de estas funciones de plegado para definir, en medida de lo posible, otras funciones generales que utilizan elementos de los tipos inductivos[14].

En este trabajo utilizamos la propiedad de que cada tipo de dato recursivo tiene un operador `fold`, el cual se abstrae gracias al uso de álgebras, y exhibimos las funciones que se pueden reescribir siguiendo este patrón y algunas que no. Esto se logra gracias a las propiedades de los operadores de plegado que dependen de la definición del tipo de dato.

1.5. Álgebra semántica

Los tipos de dato algebraicos vienen acompañados de las operaciones asociadas a él. Entre estas operaciones se encuentran las constructoras o generadoras cuyo propósito es construir un elemento del tipo de dato algebraico que se está definiendo y solo con éstas puede ser generado, es decir, no hay otra forma de construir un elemento de un tipo que no sea con las operaciones constructoras correspondientes. Para un mismo tipo de dato pueden existir diferentes operaciones constructoras con parámetros diferentes[4]. En programación funcional y en especial en Haskell a estas operaciones se les da el nombre de *constructores* de un tipo, por lo que en este trabajo les llamaremos de esta forma.

El álgebra semántica es un formalismo de especificación semántica basado en la teoría de álgebras abstractas usado para abstraer la definición de los constructores de un tipo de dato algebraico. La idea principal es analizar cada constructor con el que puede crearse un elemento de un tipo siguiendo la propia definición recursiva del tipo. De esta forma, se definen esquemas algebraicos para definir sus propiedades características[16][17][18].

Por ejemplo, se define el tipo `Nat` que representa los números naturales usando `data`, que es una primitiva en Haskell que permite definir nuevos tipos de datos algebraicos.

```
data Nat = Zero | Suc Nat
```

[4]Si el lector quiere saber más sobre el tema, revisar la referencia[15]

En este caso `Nat` tiene dos constructores, `Zero` que es un constructor constante, es decir, no recibe parámetros y `Suc` que es un constructor que recibe un argumento de tipo `Nat` y a partir de este construye un nuevo natural, el sucesor. El número 2 se representa como `Suc (Suc Zero)`.

Para definir el álgebra semántica correspondiente a `Nat` se tienen que identificar primero los tipos de sus constructores. El constructor `Zero` es de tipo `Nat`, pues `Zero` por si sólo ya es un natural. Mientras que el constructor `Suc` tiene el tipo función con dominio y codominio en `Nat`, es decir, recibe un natural y regresa un natural (`Nat ->Nat`). Con estos tipos se puede definir el álgebra con la primitiva `type`, usada para definir sinónimos[5] en Haskell, como sigue:

```
type NatAlgebra = (Nat, Nat -> Nat)
```

Se utiliza una tupla en donde cada elemento de ésta corresponde al tipo de un constructor. Sin embargo, esta es un álgebra especial pues tiene exactamente los tipos de los constructores y define la función identidad en el tipo de dato. Para poder usar el álgebra semántica en la definición de otras funciones sobre el tipo `Nat` se tiene que dar una definición mas general, es por esto que se parametriza el tipo `Nat` en la definición, resultando en la siguiente álgebra[18]:

```
type NatAlgebra n = (n, n -> n)
```

De esta forma, la variable de tipo `n` representa el tipo de regreso de la función que se quiera definir. Esto será mas evidente en capítulos posteriores cuando se use el álgebra para la definición de los operadores `fold`.

Para los tipos de datos utilizados en este trabajo, se definirá su álgebra semántica como una tupla. Cada elemento de la tupla corresponde a un constructor del tipo de dato representado como una función cuyos argumentos son los mismos que necesita el constructor para obtener un elemento del tipo de dato. Los constructores constantes serán funciones sin argumentos en el álgebra.

1.6. Definiciones de árboles

En el estudio de las estructuras de datos, una generalización de estructuras de almacenamiento que sigue de la estructura lista son las estructuras arbóreas o árboles.

En Teoría de Gráficas, un árbol es una gráfica no dirigida en donde para cualesquiera dos vértices existe un camino entre ellos, además está libre de ciclos es decir, es una gráfica conexa y acíclica [15].

Definición 1.1 (Árbol (gráfica))**.** *Un árbol es una gráfica sin ciclos y conexa.*

[5] Un sinónimo en Haskell es un nuevo nombre para un tipo ya existente.

Definición 1.2 (Árbol (estructura de datos))**.** *Un árbol es una estructura de datos no lineal en donde la información está organizada jerárquicamente en niveles.*

Existen conceptos particulares en los árboles para establecer la organización jerárquica de la información, veamos estas definiciones[19]:

Definición 1.3 (Raíz)**.** *Es el nodo inicial, es decir que no tiene antecesores.*

Definición 1.4 (Hijos)**.** *En un árbol se conoce como hijos o descendientes a los árboles siguientes en la jerarquía, es decir con menor jerarquía.*

Definición 1.5 (Hoja)**.** *Es un nodo terminal, es decir que no tiene descendientes.*

Definición 1.6 (Nivel)**.** *La distancia desde la raíz a un nodo. En particular, la raíz es el nodo de mayor jerarquía en el nivel 0.*

Definición 1.7 (Altura)**.** *La altura de un árbol es el número de aristas desde el máximo nivel a la raíz del árbol.*

Definición 1.8 (Tamaño)**.** *El tamaño de un árbol es el número de hojas.*

Definición 1.9 (Árbol n-ario)**.** *Un árbol n-ario es un árbol en donde todos los nodos tienen a lo más n descendientes.*

Un caso particular de los árboles n-arios son los árboles binarios, muy utilizados en Ciencias de la Computación.

Definición 1.10 (Árbol binario)**.** *Un árbol es binario si todos sus nodos tienen a lo más dos hijos.*

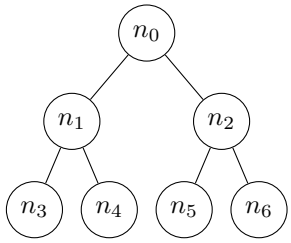


Figura 1.1: Representación gráfica de un árbol

Para ilustrar las definiciones anteriores tomemos como ejemplo el árbol de la figura 1.1 que es un árbol binario pues todos son nodos tienen a lo más dos hijos. En donde la raíz corresponde al nodo etiquetado con n_0 cuyos hijos son los nodos con las etiquetas n_1 y n_2. Las hojas son los nodos: n_3, n_4, n_5 y n_6

y éstas se encuentran en el segundo nivel. La altura de este árbol es dos y su tamaño es siete.

En este capítulo se presentaron los preliminares necesarios, así como una introducción breve de definiciones para estructuras arbóreas que servirá para el desarrollo central de este trabajo.

Capítulo 2

Listas

Las listas son una estructura de datos comúnmente usada en Ciencias de la Computación, en particular en la programación funcional por su naturaleza recursiva, la que nos permite que la implementación de éstas sea de forma muy semejante a una especificación formal.

El trabajo de programación origami sobre listas ha sido ya ampliamente desarrollado por diferentes autores [20, 1, 14, 21, 22], en este trabajo se utiliza esta estructura para dar una introducción a nociones de definiciones recursivas, álgebras semánticas, funciones de plegado y el encapsulamiento del patrón recursivo, con base en el artículo de Graham Hutton[14].

Una lista (finita) es una estructura de datos que almacena de forma linealmente ordenada elementos de un mismo tipo. Gráficamente se puede ver en la figura 2.1.

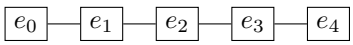


Figura 2.1: Representación gráfica de una lista de 5 elementos

Donde el primer elemento o la cabeza de la lista es e_0 y el resto o la cola de la lista es una lista con los elementos $e_1 \dots e_4$.

Enfatizamos que el orden de la lista es importante, la lista e_1, e_2, e_3 no es igual a la lista e_2, e_3, e_1.

2.1. Especificación formal

Definición 2.1 (Definición inductiva de listas)**.** *Una lista de elementos del tipo A se define recursivamente como sigue[23]:*

- *La lista vacía es una lista (*`Empty`*).*
- *Si x es un elemento de tipo A, y xs es una lista, entonces (*`Cons` x xs*) es una lista.*
- *Son todas.*

Podemos traducir la definición anterior a Haskell con el siguiente tipo de dato.

Definición 2.2 (Tipo de dato `List`)**.**

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

En donde `Empty` *es el constructor constante para la lista vacía y* `Cons` *agrega un nuevo elemento como cabeza de una lista.*

La notación en Haskell para las listas es la siguiente[9]:

- La lista vacía

  ```
  []
  ```

- La lista con cabeza `x` y cola `xs` (la operación `Cons`).

  ```
  x:xs
  ```

- En general las listas se denotan empleando corchetes y comas para separar los elementos.

  ```
  [1, 2, 3, 4]
  ```

  Esta notación es azúcar sintáctica de una sucesión del constructor `Cons` (`:`) y al final la lista vacía (`[]`), la lista anterior se puede definir también como `1:2:3:4:[]`. Esta construcción se puede ver gráficamente como un árbol como se muestra en la figura 2.2. En donde los nodos internos son la operación `:` y las hojas son los valores almacenados en la lista.

Las listas en Haskell son homogéneas, esto quiere decir que todos sus elementos son del mismo tipo. La lista `[1,2,3,4]` está bien construida pues todos los elementos son del tipo `Int`, pero la lista `[1, "hola"]` no es una lista bien formada, pues el primer elemento es de tipo `Int` mientras que el segundo es de tipo `String`.

La definición 2.1, al ser una definición recursiva, genera un principio de inducción que es útil para demostrar propiedades sobre el tipo de dato.

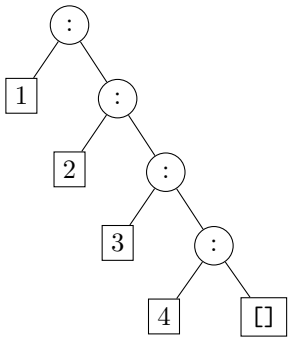


Figura 2.2: Árbol de sintaxis abstracta de la lista [1, 2, 3 , 4]

Definición 2.3 (Principio de Inducción Estructural para lista [14])**.** *Sea $\mathcal{P}$ una propiedad acerca de Listas de elementos de A. Para probar que toda lista l tiene la propiedad $\mathcal{P}$, basta demostrar lo siguiente:*

- *Base: la lista vacía tiene la propiedad $\mathcal{P}$.*
- *Hipótesis de Inducción: suponer que la propiedad se cumple para xs.*
- *Paso Inductivo: mostrar usando la hipótesis de Inducción que (Cons x xs) tiene la propiedad $\mathcal{P}$, con $x \in A$.*

Utilizando la definición 2.2, se define ahora el álgebra semántica asociada al tipo de dato `List` usando sus constructores.

Definición 2.4 (Algebra Semántica de Listas)**.** *Definición de álgebra*

```
type ListAlgebra a l = (l, a -> l -> l)
```

El álgebra semántica encapsula en un par los tipos correspondientes a los constructores: el constructor base, que no tiene argumentos y genera una Lista; y el constructor `Cons` que tiene como argumentos un elemento y una lista [1].

Muchas de las funciones definidas sobre el tipo de dato `List` siguen un patrón muy semejante al del álgebra semántica: se define un caso para tratar la lista vacía (el caso base) con una función constante, y otro caso para las listas con cabeza y cola (el caso recursivo) con una función de dos argumentos. Este patrón se puede generalizar mediante una función h y una función auxiliar $\oplus$:

```
h :: [a] -> b
h []     = e
h (x:xs) = x ⊕ h xs
```

[1]Las funciones están currificadas, es decir reciben los argumentos uno por uno[24].

La función h toma una lista, reemplaza `[]` por `e` y `(:)` por $\oplus$ [22], como se muestra en la figura 2.3. En la función `h`, `e` tiene el tipo `b` y $\oplus$ es una función con tipo `a -> b -> b`, que corresponden al patrón del álgebra semántica. El esquema de la función `h` es capturado por la función `foldList` definida a partir del álgebra semántica.

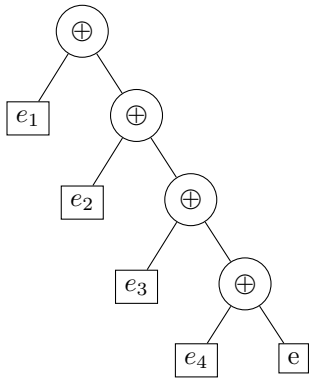


Figura 2.3: Representación gráfica de la aplicación de `h` [e_1, e_2, e_3 , e_4]

Definición 2.5 (Fold para listas)**.** *Se define la función* `foldList` *como sigue:*

```
foldList :: ListAlgebra a b -> List a -> b
foldList (e, f) = fold where
    fold Empty       = e
    fold (Cons x xs) = f x (fold xs)
```

Donde `e` es la función constante de tipo `b` y `f` es la función de dos argumentos `a -> b -> b`, descritas en la definición 2.4 del álgebra semántica. Ahora se reescribe `h` como `foldList (e,`$\oplus$`)`

Observemos que la definición 2.5 es equivalente a la definición de `foldr` para las listas que aparece en el preludio de Haskell.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ b []     = b
foldr f b (x:xs) = f x (foldr f b xs)

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b = foldList (b, f)
```

2.2. Propiedades del operador fold

El operador `fold` tiene un número importante de propiedades, para este trabajo mostramos dos de ellas: la propiedad Universal y el Principio de Fusión. Éstas son de suma importancia pues con ellas se puede generalizar cualquier otra propiedad sobre `fold`.

2.2.1. Propiedad Universal

La propiedad Universal de la función `fold` tiene su origen en la teoría de la recursividad [14].

Proposición 2.1 (Propiedad Universal de `fold` para listas [14])**.** *La propiedad Universal se puede enunciar con la siguiente equivalencia entre las definiciones de una función g sobre listas:*

$$\begin{array}{l} g\ [\,] = v \\ g\ (\mathtt{x} : \mathtt{xs}) = f\ \mathtt{x}\ (g\ \mathtt{xs}) \end{array} \quad \Longleftrightarrow \quad g = \mathtt{foldList}\ (v, f)$$

Demostración. ($\Rightarrow$ Por inducción sobre listas)

Caso Base

$$\begin{array}{cc} & g\ [\,] \\ = & \{\text{Definición de la función } g\} \\ & v \\ = & \{\text{Definición de } \mathtt{foldList}\} \\ & \mathtt{foldList}\ (v, f)\ [\,] \end{array}$$

H.I. Sea `xs` una lista, entonces $g\ \mathtt{xs} = \mathtt{foldList}\ (v, f)\ \mathtt{xs}$

Paso Inductivo

$$\begin{array}{cc} & g\ (\mathtt{x} : \mathtt{xs}) \\ = & \{\text{Definición de la función } g\} \\ & f\ x\ (g\ \mathtt{xs}) \\ = & \{\text{Hipótesis de Inducción}\} \\ & f\ \mathtt{x}\ (\mathtt{foldList}\ (v, f)\ \mathtt{xs}) \\ = & \{\text{Definición de } \mathtt{foldList}\} \\ & \mathtt{foldList}\ (v, f)\ (\mathtt{x} : \mathtt{xs}) \end{array}$$

($\Leftarrow$) Para el regreso se supone $g = \mathtt{foldList}\ (v, f)$

$$\begin{array}{cc} & \mathtt{g} \\ = & \{\text{Hipótesis}\} \\ & \mathtt{foldList}\ (v, f) \\ = & \{\text{Sustitución en la definición de } \mathtt{foldList}\} \\ & g\ [\,] = v \\ & g\ (\mathtt{x} : \mathtt{xs}) = f\ \mathtt{x}\ (g\ \mathtt{xs}) \end{array}$$

□

Intuitivamente lo que dice la Propiedad Universal del operador `foldList` es que toda función definida sobre listas con este patrón recursivo es equivalente a una instancia del operador `foldList`. Más adelante en este capítulo se dan ejemplos de funciones escritas usando `foldList` para mostrar el poder de expresividad del operador.

El uso práctico de la Propiedad Universal es como un principio de prueba. Así se puede evitar, en algunos casos, el uso de Inducción para demostrar propiedades de expresiones que usen el operador `fold` como se ve en la siguiente sección.

2.2.2. Principio de Fusión

Consideremos la siguiente ecuación entre funciones que procesan listas:

$$h \cdot \texttt{foldList}\ (w, g) = \texttt{foldList}\ (v, f)$$

Este patrón es común cuando se trata de programas que utilicen `fold` aunque no es cierto en todos los casos.

Mediante un proceso de síntesis a partir de la Propiedad Universal se obtienen las condiciones necesarias de la función h para que cumpla la ecuación anterior, como una alternativa a la inducción estructural.

1. Caso `Empty`
$$(h \cdot \texttt{foldList}\ (w, g))\ [\,] = v$$
2. Caso `Cons`
$$(h \cdot \texttt{foldList}\ (w, g))\ (\texttt{x} : \texttt{xs}) = f\ \texttt{x}\ (h \cdot \texttt{foldList}\ (w, g))\ \texttt{xs}$$

Primero se analiza el caso del constructor `Empty`

$$\begin{array}{cc} & (h \cdot \texttt{foldList}\ (w, g))\ [\,] = v \\ \Longleftrightarrow & \{\text{Composición de función}\} \\ & h\ (\texttt{foldList}\ (w, g)[\,]) = v \\ \Longleftrightarrow & \{\text{Definición de } \texttt{foldList}\} \\ & h\ w = v \end{array}$$

Ahora se hará el razonamiento correspondiente al constructor `Cons`

$$
\begin{array}{cc}
 & (h \cdot \texttt{foldList}\ (w, g))\ (\texttt{x} : \texttt{xs}) = f\ \texttt{x}\ (h \cdot \texttt{foldList}\ (w, g))\ \texttt{xs} \\
\iff & \{\text{Composición de función}\} \\
 & h\ (\texttt{foldList}\ (w, g)\ (\texttt{x} : \texttt{xs})) = f\ \texttt{x}\ (h\ (\texttt{foldList}\ (w, g)\ \texttt{xs})) \\
\iff & \{\text{Definición de } \texttt{foldList}\} \\
 & h\ (g\ \texttt{x}\ (\texttt{foldList}\ (w, g)\ \texttt{xs})) = f\ \texttt{x}\ (h\ (\texttt{foldList}\ (w, g)\ \texttt{xs})) \\
 & \{\text{Renombramos } (\texttt{foldList}\ (w, g)\ \texttt{xs}) = y\} \\
 & h\ (g\ \texttt{x}\ y) = f\ \texttt{x}\ (h\ y)
\end{array}
$$

Así obtenemos las condiciones

1. $h\ w = v$
2. $h\ (g\ \texttt{x}\ y) = f\ \texttt{x}\ (h\ y)$

que juntas son suficientes para garantizar el principio de Fusión del operador `foldList` para listas.

Definición 2.6 (Función estricta)**.** *una función estricta es aquella que si se aplica a un argumento indefinido, regresa indefinido, es decir, si f es una función estricta, entonces es cierto que $f(\texttt{undef}) = \texttt{undef}$.*

Lema 2.1 (Principio de Fusión para listas [14])**.** *Sea h una función tal que cumple las siguientes condiciones*

- $h\ w = v$
- $h\ (g\ \texttt{x}\ y) = f\ \texttt{x}\ (h\ y)$

con f una función estricta. Entonces para la composición de h con el operador `foldList` *de listas es cierto que*

$$h \cdot \texttt{foldList}\ (w, g) = \texttt{foldList}\ (v, f)$$

El principio anterior permite que la composición de funciones, usando la función `fold`, se traduzca en la aplicación de un `fold` con una función de paso más compleja. De esta forma se puede mejorar la eficiencia en tiempo de los programas al componer ambas funciones en una sola llamada al operador `foldList` y así recorrer una única vez la lista.

2.3. Ejemplos

En esta sección se muestran algunos ejemplos de funciones definidas con el patrón mostrado en las secciones anteriores y su traducción a instancias del

operador `fold` así como ejemplos de funciones que no es posible reescribir con `fold` junto con una justificación de este comportamiento.

2.3.1. Funciones con patrón de reescritura con `fold`

Usando `fold` se puede definir cualquier función sobre listas que siga el patrón que encapsula es decir, al definir un comportamiento para el caso base y uno recursivo para el constructor `Cons`. A continuación se muestran ejemplos de estas funciones mostrando primero la implementación recursiva *natural* y luego la traducción con el operador `fold`. Para simplificar las definiciones se usará notación de listas de Haskell.

concat Define la concatenación de dos listas

```
concat :: [[a]] -> [a]
concat []     = []
concat (x:xs) = x ++ (concat xs)

concat :: [[a]] -> [a]
concat = foldList ([], (++))
```

reverse Calcula la reversa de una lista

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = (reverse xs) ++ [x]

reverse :: [a] -> [a]
reverse = foldList ([], (\x xs -> xs ++ [x]))
```

length Calcula el número de elementos de una lista

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 ++ (legth xs)

length :: [[a]] -> [a]
length = foldList (0, (\x y -> 1 + y))
```

sum Suma todos los elementos de una lista de valores numéricos

```
sum :: (Num a) => [a] -> Int
sum []     = 0
sum (x:xs) = x + (sum xs)

sum :: (Num a) => [a] -> Int
sum = foldList (0, (\x y -> x + y))
```

and Aplica la conjunción lógica sobre todos los elementos de una lista de Booleanos

```
and :: [Bool] -> Bool
and []     = True
and (x:xs) = x && (and xs)

and :: [Bool] -> Bool
and = foldList (True, (&&))
```

2.3.2. Funciones que no se pueden reescribir con fold

Para ejemplificar las funciones que no se pueden traducir utilizando `foldList` veremos la función `foldl` que se define en Haskell como sigue:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ a []     = a
foldl f a (x:xs) = foldl f (f a x) xs
```

Se puede observar que esta función se implementa con el patrón que define un comportamiento para el caso de la lista vacía y uno para el caso del constructor `cons`, sin embargo cuando se define el comportamiento para el caso de `cons` no se utiliza el patrón `h (x:xs) = x` $\oplus$ `h xs` que es el que encapsula el operador `foldList` sino que la llamada recursiva se realiza en primera instancia. Es por esto que la función `foldl` no puede reescribirse como instancia de `foldList`.

Otro ejemplo es la función `shift` que hace una rotación sobre la lista, por ejemplo `shift [1,2,3] = [2,3,1]`, definida de la siguiente forma:

```
shift :: [a] -> [a]
shift []     = []
shift (x:xs) = xs ++ [x]
```

Esta función no puede traducirse a una instancia de `foldList` pues no se trata de una función recursiva, `foldList` es una función que itera sobre toda la lista y la función `shift` sólo opera con la cabeza de la lista sin iterar sobre la cola.

2.3.3. Ejemplos avanzados

En esta sección se verán ejemplos de funciones mas complejas que entran en el patrón de reescritura del operador `foldList`

map Función de orden superior que aplica una función (que recibe como argumento) a todos los elementos de una lista.

```
map ::(a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = (f x) :(map f xs)

map :: (a -> b) -> [a] -> [b]
map f = foldList ([], (f.cons))
   where cons x xs = x:xs
```

filter Función de orden superior que recibe un predicado[2] y elimina de la lista aquellos elementos que no lo cumplan.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []     = []
filter f (x:xs)
 | (f x)        = x:fxs
 | otherwise    = fxs
where fxs = filter f xs

filter :: (a -> Bool) -> [a] -> [a]
filter f = foldList ([], (fil)) where
   fil x xs = if f x then x:xs else xs
```

La ventaja de traducir todas estas funciones a instancias de `foldList` es que se pueden usar las propiedades que vimos anteriormente para este operador. En especial el uso del principio de fusión, con el cual podemos combinar estas funciones en una sola llamada a `foldList`, lo que se traduce a recorrer la lista una única vez, y eso representa una mejora significativa en eficiencia en tiempo sobre la composición de funciones.

2.4. Unfold

En lo que va de este capítulo, se expusieron los beneficios de encapsular patrones comunes de funciones recursivas en operadores de orden superior cuando

[2]Un predicado es una función que regresa un valor de verdad, es decir, una función tiene como codominio el tipo `Bool`.

se definen funciones sobre un tipo de dato inductivo. De igual modo existe un operador dual a `fold`, la función `unfold` definida a continuación [25].

Definición 2.7 (Unfold de Listas)**.** *El operador unfold para las listas construidas en este capítulo se define en* Haskell *como sigue:*

```
unfoldList :: (b -> Maybe(a,b)) -> b -> List a
unfoldList f b = case (f b) of
   Just (x, xs) -> Cons x (unfoldList f xs)
   Nothing      -> Empty
```

Este operador es considerado dual pues, en un sentido no estricto, deshace el efecto de `fold` sobre una lista, es decir, reconstruye la lista a partir de la cual se obtuvo un valor usando funciones de plegado. Se expresa esta noción con la siguiente ecuación:

$$\texttt{unfoldList}\ f\ (\texttt{foldList}\ (v, g)\ xs) = xs$$

Es importante observar que para que la ecuación anterior sea cierta debe existir una relación entre las funciones f y g, definida como $f\ (g\ a\ b) = (a, b)$. Debido a que no hay forma de garantizar esta restricción se utiliza el tipo `Maybe` para tratar los errores: si se llega a `Nothing` entonces se concluye que la expresión fue construida con un caso base, es decir la lista vacía.

En contraste con `fold`, el operador `unfold` es menos estudiado y utilizado pues su poder de expresividad es menor y mas difícil de abstraer [25].

Para ejemplificar el uso de la función `unfold` vamos definir una lista que tenga a todos los números de Fibonacci utilizando la evaluación perezosa de Haskell para definir una lista infinita, de la siguiente forma:

```
fibs :: [Integer]
fibs unfoldList
      (\(f0, f1) -> Just (f1, (f1, f0+f1))) (0, 1)
```

En este ejemplo siempre tenemos el número de Fibonacci actual y el anterior en una tupla `(actual, anterior)`, en cada iteración agregamos el actual a la lista y para la siguiente iteración los valores de la tupla son `(actual+anterior ,actual)` de esta forma el valor actual se vuelve el anterior en la siguiente iteración y el actual es la suma de ambos.

El comportamiento de `unfold` se puede ver como un ciclo `forEach` que va iterando sobre una lista [26]. Recordemos que `unfold` construye una lista a partir de un valor, de esta forma se puede definir un ciclo `forEach` como sigue.

```
forEach f = unfoldList (\x -> Just (x, f x))
```

Con esta definición `forEach succ 0` es la lista que contiene todos los números naturales, es decir se va iterando sobre una lista infinita.

En este capítulo se introducen conceptos relacionados con la definición de estructuras de datos, así como el uso tradicional de operadores de plegado y sus propiedades para el tipo de dato Lista.

Capítulo 3

Árboles binarios con información sólo en las hojas (tipo 1)

Esta estructura de datos es un caso particular de árboles binarios en donde los únicos nodos que almacenan información son las hojas o nodos terminales, los nodos internos no almacenan datos[2].

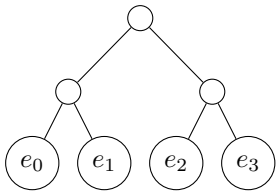


Figura 3.1: Representación gráfica de un árbol de tipo 1

3.1. Especificación formal

Definición 3.1 (Definición inductiva de árboles binarios con información sólo en las hojas)**.** *Un árbol de elementos de tipo A se define como sigue:*

- *Si x es un elemento de tipo A entonces (*`Leaf` *x) es un árbol.*
- *Si t_1 y t_2 son árboles entonces (*`Node` *t_1 t_2) es un árbol.*
- *Son todos*

De la definición anterior, el constructor `Leaf` representa a una hoja que almacena al elemento x y el constructor `Node` representa al árbol que tiene como hijo izquierdo a t_1 y como hijo derecho a t_2 los cuales son árboles bien formados.

La definición 3.1 se traduce a Haskell con el tipo de dato algebraico `Tree`.

Definición 3.2 (Tipo de dato `Tree`)**.**
```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Estos árboles solo tienen información en las hojas pues el constructor `Leaf` es el único que almacena elementos de tipo `a`.

El principio de inducción generado por la definición 3.1 servirá más adelante para demostrar propiedades sobre este tipo.

Definición 3.3 (Principio de Inducción Estructural para árboles con información solo en las hojas)**.** *Sea $\mathcal{P}$ una propiedad acerca de árboles de elementos de A. Para probar que todo árbol cumple la propiedad $\mathcal{P}$, basta demostrar lo siguiente:*

- *Base: el constructor (*`Leaf` *x) cumple $\mathcal{P}$ con $x \in A$.*
- *Hipótesis de Inducción: suponer que $\mathcal{P}$ se cumple para los árboles t_1 y t_2 respectivamente.*
- *Paso Inductivo: mostrar usando la hipótesis de inducción que (*`Node` *t_1 t_2) cumple $\mathcal{P}$.*

A continuación daremos la definición en Haskell del álgebra semántica asociada al tipo `Tree`.

Definición 3.4 (Álgebra semántica del tipo `Tree`)**.**
```
type TreeAlgebra x t = (x -> t, t -> t -> t)
```

Veamos ahora cuál es el patrón que van a seguir algunas funciones definidas para el tipo de dato algebraico `Tree`.

```
h (Leaf x)   = f x
h (Node l r) = g (h l) (h r)
```

Observemos que `h` toma un árbol y reemplaza `Leaf` por `f` y `Node` por `g` aplicando recursivamente `h` a los subárboles del nodo. Este esquema es capturado por la función `foldTree` definida usando el álgebra semántica 3.4. Nuevamente, observemos que se trata de un árbol binario homogéneo pues todos sus elementos son del tipo `a`.

Definición 3.5 (Fold General de Tree)**.** *Se define la función* `foldTree` *como sigue:*

```
foldTree :: TreeAlgebra x t -> Tree x -> t
foldTree (f, g) = fold where
  fold (Leaf x)   = f x
  fold (Node l r) = g (fold l) (fold r)
```

Reescribimos `h` en términos de `foldTree`, de la siguiente forma `h = foldTree (f,g)`. Encapsulando así el patrón de definición de funciones para el tipo `Tree` con la función `foldTree`.

3.2. Propiedades del operador fold

En esta sección se estudian las propiedades vistas anteriormente (en el caso de Listas y la función `foldList`) para la función `foldTree` de árboles con información en las hojas.

3.2.1. Propiedad Universal

Comenzamos por definir la unicidad de `foldTree`.

Proposición 3.1 (Propiedad Universal de foldTree)**.** *Para cualquier función g se cumple que:*

$$\begin{array}{l} g\ (\texttt{Leaf}\ x) = f\ x \\ g\ (\texttt{Node}\ t_1\ t_1) = h\ (g\ t_1)\ (g\ t_2) \end{array} \quad \Longleftrightarrow \quad g = \texttt{foldTree}\ (f, h)$$

Demostración. ($\Rightarrow$ Por inducción sobre árboles)

Caso Base

$$\begin{array}{cc} & g\ (\texttt{Leaf}\ x) \\ = & \{\text{Definición de la función } g\} \\ & f\ x \\ = & \{\text{Definición de } \texttt{foldTree}\} \\ & \texttt{foldTree}\ (f, h)\ (\texttt{Leaf}\ x) \end{array}$$

H.I. Sean t_1 y t_2 árboles, entonces $g\ t_1 = \texttt{foldTree}(f, h)\ t_1$ y $g\ t_2 = \texttt{foldTree}(f, h)\ t_2$.

Paso Inductivo

$$\begin{array}{cc} & g\ (\texttt{Node}\ t_1\ t_2) \\ = & \{\text{Definición de la función } g\} \\ & h\ (g\ t_1)\ (g\ t_2) \\ = & \{\text{Hipótesis de Inducción}\} \\ & h\ (\texttt{foldTree}\ (f,h)\ t_1)\ (\texttt{foldTree}\ (f,h)\ t_2) \\ = & \{\text{Definición de } \texttt{foldTree}\} \\ & \texttt{foldTree}\ (f,h)\ (\texttt{Node}\ t_1\ t_2) \end{array}$$

($\Leftarrow$)

$$\begin{array}{cc} & g \\ = & \{\text{Hipótesis}\} \\ & \texttt{foldTree}\ (f,h) \\ \Rightarrow & \{\text{Sustituyendo en la definición de } \texttt{foldTree}\} \\ & g\ (\texttt{Leaf}\ x) = f\ x \\ & g\ (\texttt{Node}\ t_1\ t_2) = h\ (g\ t_1)\ (g\ t_2) \end{array}$$

□

Igual que en el caso de listas la Propiedad Universal nos dice que cualquier función definida para árboles usando el patrón indicado, se puede reescribir utilizando el operador `foldTree`.

3.2.2. Principio de Fusión

Considerando la siguiente ecuación

$$h \cdot \texttt{foldTree}\ (g,w) = \texttt{foldTree}\ (f_1,f_2)$$

y como lo hicimos en el caso de listas hacemos un proceso de síntesis usando la Propiedad Universal para encontrar las restricciones sobre la función h y poder establecer el Principio de Fusión:

Aplicando la Propiedad Universal correspondiente se tienen las siguientes ecuaciones:

1. Caso Leaf:
$$h \cdot \texttt{foldTree}\ (g,w)\ (\texttt{Leaf}\ e) = f_1\ e$$
2. Caso Node:
$$\begin{array}{cc} & h \cdot \texttt{foldTree}\ (g,w)\ (\texttt{Node}\ t_1\ t_2) \\ = & f_2\ (h \cdot \texttt{foldTree}\ (g,w)\ t_1)\ (h \cdot \texttt{foldTree}\ (g,w)\ t_2) \end{array}$$

Desarrollemos primero la ecuación que corresponde al caso de `Leaf`.

$$
\begin{array}{cc}
 & h \cdot \texttt{foldTree}\ (g, w)\ (\texttt{Leaf}\ e) = f_1\ e \\
\iff & \{\text{Definición de Composición}\} \\
 & h\ (\texttt{foldTree}\ (g, w)\ (\texttt{Leaf}\ e)) = f_1\ e \\
\iff & \{\text{Definición de } \texttt{foldTree}\} \\
 & h\ (g\ e) = f_1\ e \\
\iff & \{\text{Definición de Composición}\} \\
 & h \cdot g = f_1
\end{array}
$$

Ahora hagamos el proceso de síntesis sobre la ecuación que corresponde al constructor `Node`

$$
\begin{array}{cc}
 & h \cdot \texttt{foldTree}\ (g, w)\ (\texttt{Node}\ t_1\ t_2) \\
= & f_2\ (h \cdot \texttt{foldTree}\ (g, w)\ t_1)\ (h \cdot \texttt{foldTree}\ (g, w)\ t_2) \\
\iff & \{\text{Definición de composición}\} \\
 & h\ (\texttt{foldTree}\ (g, w)\ (\texttt{Node}\ t_1\ t_2)) \\
= & f_2\ (h\ (\texttt{foldTree}\ (g, w)\ t_1))\ (h\ (\texttt{foldTree}\ (g, w)\ t_2)) \\
\iff & \{\text{Definición de } \texttt{foldTree}\} \\
 & h\ (w\ (\texttt{foldTree}\ (g, w)\ t_1)\ (\texttt{foldTree}\ (g, w)\ t_2)) \\
= & f_2\ (h\ (\texttt{foldTree}\ (g, w)\ t_1))\ (h\ (\texttt{foldTree}\ (g, w)\ t_2)) \\
\iff & \{\text{Renombrando } y_1 = \texttt{foldTree}\ (g, w)\ t_1 \text{ y } y_2 = \texttt{foldTree}\ (g, w)\ t_2\} \\
 & h\ (w\ y_1\ y_2) = f_2\ (h\ y_1)\ (h\ y_2)
\end{array}
$$

Con lo que podemos concluir que la propiedad de fusión del operador `foldTree` es la siguiente:

Lema 3.1 (Principio de Fusión para `foldTree`)**.** *Sea h una función tal que cumple las siguientes restricciones*

- $h \cdot g = f_1$
- $h\ (w\ y_1\ y_2) = f_2\ (h\ y_1)\ (h\ y_2)$

con f_2 una función estricta. Entonces es cierto que

$$h \cdot \texttt{foldTree}\ (g, w) = \texttt{foldTree}\ (f_1, f_2)$$

3.3. Ejemplos

En esta sección se muestran algunos ejemplos de traducción de funciones recursivas con el operador `foldTree`.

3.3.1. Funciones con patrón de reescritura con fold

tam Regresa el tamaño de un árbol, donde el tamaño es el número de hojas.

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

tamf :: Tree a -> Int
tamf = foldTree ((\x -> 1),(+))
```

alt Regresa la altura de un árbol, donde la altura es el número de niveles en el árbol.

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

altf :: Tree a -> Int
altf = foldTree (\x -> 0,\x y -> (max x y) + 1)
```

aplana Convierte el árbol en una lista.

```
aplana :: Tree a -> [a]
aplana (Leaf e)     = [e]
aplana (Node t1 t2) = (aplanar t1) ++ (aplanar t2)

aplanaf :: Tree a -> [a]
aplanaf = foldTree ((\x -> [x]), (\x y -> x ++ y))
```

nodos Regresa el número de nodos del árbol.

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

nodosf :: Tree a -> Int
nodosf = foldTree ((\x -> 1), (\x y -> 1 + x + y))
```

3.3.2. Ejemplos Avanzados

mapTree Generalización de la función `map` para la estructura `Tree`.

```
mapT :: Tree a -> (a -> b) -> Tree b
mapT (Leaf e) f   = Leaf (f e)
mapT (Node l r) f = Node (mapT l f) (mapT r f)

mapTf :: (a -> b) -> Tree a -> Tree b
mapTf f = foldTree
            (\x -> Leaf (f x), \x y -> Node x y)
```

sub Regresa una lista con todos los subárboles propios contenidos.

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

subf :: Tree a -> [Tree a]
subf = foldTree
         (\x -> [Leaf x],
          \x y ->[Node (head x)(head y)]++ x ++y)
```

3.4. Unfold

Para poder definir la función `unFoldT` se define el tipo de dato algebraico `MaybeT` para que los constructores funcionen a nuestra conveniencia, es decir en donde cada constructor en `MaybeT` le corresponde uno de los casos de la definición de `Tree`.

Definición 3.6 (`MaybeT`)**.** *Definimos el nuevo tipo* `MaybeT` *en* Haskell

```
data MaybeT b = One b | Both (b, b)
```

Con lo anterior se puede implementar `unfold` en Haskell de la siguiente forma:

Definición 3.7 (Unfold Tree)**.** *La función* `unfoldT` *se define como:*

```
unfoldT::(b -> MaybeT b) -> (b -> a) -> b -> Tree a
unfoldT f g b = case (f b) of
   One lf     -> Leaf (g lf)
   Both (l,r) ->
      Node (unfoldT f g l) (unfoldT f g r)
```

Una observación importante acerca de la función `unfoldT` es que no hay forma de garantizar que el árbol construido a partir de un valor sea el mismo con el que se generó este valor, es decir, la ecuación

$$\texttt{unfoldT}\ f\ g\ (\texttt{foldTree}\ (g', f')\ t) = t$$

no es cierta en todos los casos. El árbol resultante dependerá completamente de las funciones f y g que se reciban como argumentos. Es por esto que la operación de despliegue no es sencilla de abstraer.

Por ejemplo si se dobla un árbol con la operación `(+)`, es decir se calcula la suma de los elementos, existen una infinidad de árboles a partir de los cuales se puede obtener el mismo resultado, una forma de hacerlo es agregar hojas que almacenan el valor `0` que no afecta la suma. Si se tienen los siguientes árboles:

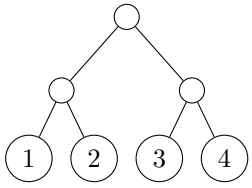


Figura 3.2: `t1`

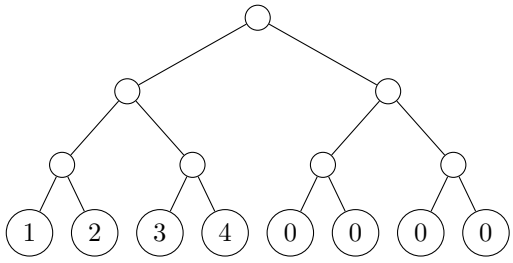


Figura 3.3: `t2`

Correspondientes en Haskell a:

```
t1 :: Tree Int
t1 =  (Node
       (Node (Leaf 1) (Leaf 2))
       (Node (Leaf 3) (Leaf 4)))

t2 :: Tree Int
t2 = (Node
```

```
(Node
   (Node (Leaf 1) (Leaf 2))
   (Node (Leaf 3) (Leaf 4)))
(Node
   (Node (Leaf 0) (Leaf 0))
   (Node (Leaf 0) (Leaf 0))))
```

Al aplicar la función `foldTree((+),id)`sobre cada uno de los árboles, en donde `id` es la función identidad[1] ya definida en Haskell, se obtiene.

```
foldTree ((+),id) t1 = 10

foldTree ((+),id) t2 = 10
```

El resultado es el mismo a pesar de que estructuralmente los árboles son diferentes, y ambos árboles podrían ser el resultado de `unfoldT f id 10`, dependiendo de cómo se defina la función `f`.

[1] La función que regresa el mismo argumento que recibe.

Capítulo 4

Árboles binarios con información en todos los nodos (tipo 2)

En este capítulo se estudia una estructura arbórea más general, estos árboles son también binarios pero a diferencia de los vistos en el capítulo anterior almacenan información no sólo en las hojas sino también en los nodos internos[2] y gráficamente se pueden representar como se ve en la figura 4.1.

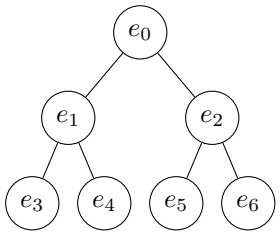


Figura 4.1: Representación gráfica de un árbol de tipo 2

4.1. Especificación formal

Definición 4.1 (Definición inductiva de árboles binarios con información en todos los nodos)**.** *Un árbol de elementos de tipo A se define como sigue[23]:*

- `Void` *es un árbol binario.*

- *Si x es de tipo A y t_1 y t_2 son árboles binarios entonces (*`Node` x t_1 t_2*) es un árbol binario.*
- *Son todos*

De esta definición el constructor `Void` representa el árbol vacío, necesario como caso base de la estructura recursiva, mientras que `Node` construye un árbol a partir de una raíz x, un subárbol izquierdo t_1 y un subárbol derecho t_2. Este árbol es una estructura homogénea pues solo se almacenan elementos del tipo A.

La definición 4.1 resulta en la siguiente implementación en Haskell:

Definición 4.2 (Tipo de dato `BinT`)**.**

```
data BinT a = Void | Node a (BinT a) (BinT a)
```

El principio de inducción generado por la definición 4.1 es el siguiente:

Definición 4.3 (Principio de Inducción Estructural para el tipo de dato `BinT`)**.** *Sea $\mathcal{P}$ una propiedad acerca de Árboles de elementos de A. Para probar que para todo árbol $t \in BinT$ cumple la propiedad $\mathcal{P}$, basta demostrar lo siguiente:*

- *Base: el constructor* `Void` *cumple $\mathcal{P}$.*
- *Hipótesis de Inducción: suponer que $\mathcal{P}$ se cumple para los árboles t_1 y t_2.*
- *Paso Inductivo: mostrar usando la hipótesis de Inducción que (*`Node` x t_1 t_2*) cumple $\mathcal{P}$ con x de tipo A.*

Ahora se definirá el álgebra semántica de `BinT` en Haskell, para definir el operador `fold` correspondiente.

Definición 4.4 (Álgebra Semántica de `BinT`)**.** *Definición de álgebra*

```
type BinAlgebra x t = (t, x -> t -> t -> t)
```

Recordemos que el par que conforma al álgebra son los tipos correspondientes a los constructores de `BinT`.

A continuación se define el patrón general que siguen algunas funciones definidas para `BinT` en términos de una función `h`.

```
h Void          =  v
h (Node x l r) = n x (h l) (h r)
```

Este es el esquema capturado con el operador `foldBin` definido a continuación.

Definición 4.5 (Fold General de `BinT`)**.** *Se define la función* `foldBin` *como sigue:*

```
foldBin :: BinAlgebra x t -> BinT x -> t
foldBin (b, f) = fold where
  fold Void         = b
  fold (Node x l r) = f x (fold l) (fold r)
```

Reescribimos `h` en términos de `foldBin` 4.5, de la siguiente forma `h = foldBin(v,n)` encapsulando así el patrón de definición de funciones para el tipo `BinT` con la función `foldBin`.

4.2. Propiedades del operador foldBin

En esta sección se definen la Propiedad Universal y el Principio de Fusión para `BinT`.

4.2.1. Propiedad Universal

Proposición 4.1 (Propiedad Universal de foldBin)**.** *Para cualquier función g se cumple que:*

$$g\ \texttt{Void} = v$$
$$g\ (\texttt{Node}\ x\ t_1\ t_1) = n\ x\ (g\ t_1)\ (g\ t_2) \quad \Longleftrightarrow \quad g = \texttt{foldBin}\ (v, n)$$

Demostración. ($\Rightarrow$ Utilizando el principio de inducción para BinT 4.3)

Caso Base

$$\begin{array}{cc} & g\ \texttt{Void} \\ = & \{\text{Definición de la función } g\} \\ & v \\ = & \{\text{Definición de } \texttt{foldBin}\} \\ & \texttt{foldBin}\ (v, n)\ \texttt{Void} \end{array}$$

H.I. Sean t_1 y t_2 árboles, entonces $g\ t_1 = \texttt{foldBin}\ (v, n)\ t_1$ y $g\ t_2 = \texttt{foldBin}\ (v, n)\ t_2$.

Paso Inductivo

$$\begin{array}{cc} & g\ (\texttt{Node}\ x\ t_1\ t_2) \\ = & \{\text{Definición de la función } g\} \\ & n\ x\ (g\ t_1)\ (g\ t_2) \\ = & \{\text{Hipótesis de Inducción}\} \\ & n\ x\ (\texttt{foldBin}\ (v, n)\ t_1)\ (\texttt{foldBin}\ (v, n)\ t_2) \\ = & \{\text{Definición de } \texttt{foldBin}\} \\ & \texttt{foldBin}\ (v, n)\ (\texttt{Node}\ x\ t_1\ t_2) \end{array}$$

($\Leftarrow$)

$$
\begin{array}{rc}
 & g \\
= & \{\text{Hipótesis}\} \\
 & \texttt{foldBin}\ (v, n) \\
\Rightarrow & \{\text{Sustituyendo en la definición de } \texttt{foldBin}\} \\
 & g\ \texttt{Void} = v \\
 & g\ (\texttt{Node}\ x\ t_1\ t_2) = n\ x\ (g\ t_1)\ (g\ t_2)
\end{array}
$$

□

Con esto ha quedado demostrado que cualquier función definida para la estructura `binT` siguiendo el esquema definido con la función h se puede reescribir utilizando el operador `foldBin`.

4.2.2. Principio de Fusión

A partir de la siguiente ecuación:

$$h \cdot \texttt{foldBin}\ (g, w) = \texttt{foldBin}\ (v, n)$$

se usa un proceso de síntesis con la Propiedad Universal para encontrar las restricciones sobre la función `h` para que esta ecuación se cumpla.

Aplicando la Propiedad Universal correspondiente se tienen las siguientes ecuaciones:

1. Caso Void:

$$h \cdot \texttt{foldBin}\ (w, g)\ \texttt{Void} = v$$

2. Caso Node:

$$
\begin{array}{rc}
 & h \cdot \texttt{foldBin}\ (w, g)\ (\texttt{Node}\ x\ t_1\ t_2) \\
= & n\ x\ (h \cdot \texttt{foldBin}\ (w, g)\ t_1)\ (h \cdot \texttt{foldBin}\ (w, g)\ t_2)
\end{array}
$$

Se resuelve primero la ecuación que corresponde al caso de `Void`.

$$
\begin{array}{rc}
 & h \cdot \texttt{foldBin}\ (w, g)\ \texttt{Void} = v \\
\Longleftrightarrow & \{\text{Definición de Composición}\} \\
 & h\ (\texttt{foldBin}\ (w, g)\ \texttt{Void}) = v \\
\Longleftrightarrow & \{\text{Definición de } \texttt{foldBin}\} \\
 & h\ w = v
\end{array}
$$

Ahora se desarrolla la ecuación que corresponde al constructor `Node`:

$$
\begin{array}{rc}
 & h \cdot \texttt{foldBin}\ (w, g)\ (\texttt{Node}\ x\ t_1\ t_2) \\
= & n\ x\ (h \cdot \texttt{foldBin}\ (w, g)\ t_1)\ (h \cdot \texttt{foldBin}\ (w, g)\ t_2) \\
\iff & \{\text{Definición de composición}\} \\
 & h\ (\texttt{foldBin}\ (w, g)\ (\texttt{Node}\ x\ t_1\ t_2)) \\
= & n\ x\ (h\ (\texttt{foldBin}\ (w, g)\ t_1))\ (h\ (\texttt{foldBin}\ (w, g)\ t_2)) \\
\iff & \{\text{Definición de } \texttt{foldBin}\} \\
 & h\ (g\ x\ (\texttt{foldBin}\ (w, g)\ t_1)\ (\texttt{foldBin}\ (w, g)\ t_2)) \\
= & n\ x\ (h\ (\texttt{foldBin}\ (w, g)\ t_1))\ (h\ (\texttt{foldBin}\ (w, g)\ t_2)) \\
\iff & \{\text{Renombrando } y_1 = \texttt{foldBin}\ (w, g)\ t_1 \text{ y } y_2 = \texttt{foldBin}\ (w, g)\ t_2\} \\
 & h\ (g\ x\ y_1\ y_2) = n\ x\ (h\ y_1)\ (h\ y_2)
\end{array}
$$

Después de encontrar las restricciones sobre la función h, se define el Principio de Fusión

Lema 4.1 (Principio de Fusión para `BTree`)**.** *Sea h una función tal que cumple las siguientes condiciones*

- $h\ w = v$
- $h\ (g\ x\ y_1\ y_2) = n\ x\ (h\ y_1)\ (h\ y_2)$

con n una función estricta, entonces es cierto que

$$h \cdot \texttt{foldBin}\ (w, g) = \texttt{foldBin}\ (v, n)$$

4.3. Ejemplos

Ahora se dan ejemplos de funciones definidas sobre `BinT` con el patrón encapsulado por el operador `foldBin` y su reescritura con la técnica de programación origami.

4.3.1. Funciones con patrón de reescritura con fold

Comenzamos definiendo funciones sencillas para árboles con información en todos los nodos.

inorder Regresa en forma de lista los elementos del árbol siguiendo el recorrido *in order*

```
inorder :: BinT a -> [a]
inorder Void         = []
inorder (Node x l r) =(inorder l) ++ x:(inorder r)

inorderF :: BinT a -> [a]
inorderF = foldBin ([], (\x l r -> l++ x:r))
```

preorder Regresa en forma de lista los elementos del árbol siguiendo el recorrido *pre order*

```
preorder :: BinT a -> [a]
preorder Void         = []
preorder (Node x l r) = x:(preorder l) ++
                        (preorder r)

preorderF :: BinT a -> [a]
preorderF = foldBin ([],(\x l r -> x:l++r))
```

postorder Regresa en forma de lista los elementos del árbol siguiendo el recorrido *post order*

```
postorder :: BinT a -> [a]
postorder Void         = []
postorder (Node x l r) = (postorder l) ++
                         (postorder r) ++
                         [x]

postorderF :: BinT a -> [a]
postorderF = foldBin ([], (\x l r -> l++r++[x]))
```

nnodos Regresa el número de nodos de un árbol.

```
nnodos :: BinT a -> Int
nnodos Void         = 0
nnodos (Node _ l r) = 1 + (nnodos l) + (nnodos r)

nnodosF :: BinT a -> Int
nnodosF = foldBin (0, \x l r -> 1+l+r)
```

elem Verifica si un elemento pertenece a un árbol.

```
elem :: (Eq a) => a -> BinT a -> Bool
elem _ Void         = False
elem e (Node x l r) = (x == e) ||
                      (elem e l) ||
                      (elem e r)

elemF :: (Eq a) => a -> BinT a -> Bool
elemF e = foldBin (False,
                   (\x l r -> (e == x) || l || r))
```

4.3.2. Funciones que no se pueden reescribir con fold

Veamos un ejemplo de una función que no puede reescribirse en programación origami: `remove` que elimina un elemento de un árbol.

```
remove :: (Eq a) => a -> BinT a -> BinT a
remove _ Void              = Void
remove e t@(Node x l r)
 | (e == x) && (isvoid l) = r
 | (e == x) && (isvoid r) = l
 | (e == x)               = (Node rr l nr)
 | (elem e l)             = (Node x le r)
 | (elem e r)             = (Node x l re)
 | otherwise              = t
 where rr = root r
       nr = remove rr r
       le = remove e l
       re = remove e r
```

En donde el operador `@` en Haskell significa *Leer como* y es utilizado para asignarle un nombre al patrón que está cazando mientras que la función `isvoid` verifica si un árbol es vacío y `root` regresa la raíz de un árbol, estas funciones están definidas de la siguiente manera:

```
void :: BinT a -> Bool
void Void = True
void _    = False

root :: BinT a -> a
root (Node _ x _) = x
```

La función `remove` en el caso en los árboles que son construidos usando el constructor `Node`, se deben verificar varios casos posibles sobre la estructura del

árbol; sin embargo el patrón encapsulado por el operador `foldBin` no permite hacerlo por lo que no es posible reescribir `remove` siguiendo este patrón pues la verificación de estos casos es necesaria.

4.3.3. Ejemplos Avanzados

mapBT Función `map` para la estructura `BinT`

```
mapBT :: (a -> b) -> BinT a -> BinT b
mapBT _ Void          = Void
mapBT f (Node x l r) = Node (f x)
                            (mapBT f l)
                            (mapBT f r)

mapBTF :: (a -> b) -> BinT a -> BinT b
mapBTF f = foldBin (Void,
                    (\x l r -> Node (f x) l r))
```

filterBT Función `filter` para la estructura `BinT`

```
filterBT :: (a -> Bool) -> BinT a -> BinT a
filterBT _ Void          = Void
filterBT f (Node x l r)
 | f x                   = nw
 | otherwise             = remove x nw
where fl = (filterBT f l)
      fr = (filterBT f r)
      nw = Node x fl fr

filterBTF :: (a -> Bool) -> BinT a -> BinT a
filterBTF f = foldBin (Void, g)
 where g x l r = if f x then nw else remove x nw
       fl      = (filterBT f l)
       fr      = (filterBT f r)
       nw      = Node x fl fr))
```

En este ejemplo es importante observar que a pesar de que la función de `filterBTF` si está definida utilizando el operador `foldBin` no se trata de una función en programación origami *pura* pues depende de `remove` que como ya vimos no puede reescribirse con `foldBin`

4.4. Unfold

La función `unfoldB` que construye un árbol a partir de una función de *desdoble* y un valor se define como sigue:

Definición 4.6 (Unfold para `BinT`). *El operador unfold para los árboles con información en todos los nodos se define en* Haskell *como sigue:*

```
unfoldB :: (b -> Maybe(a,b,b)) -> b -> BinT a
unfoldB f b = case (f b) of
 Just(x,l,r) -> Node (unfoldB f l) x (unfoldB f r)
 Nothing     -> Void
```

En este caso la función usada para construir el árbol, es una función que recibe un argumento de tipo `b` y regresa un `Maybe (a,b,b)` en donde la tupla representa los parámetros necesarios para construir un árbol binario, de la misma forma en la que se abstrajo para la definición del álgebra semántica.

De la misma forma que el operador `unfold` para árboles con información solo en las hojas definido en el capítulo anterior, este operador no siempre satisface la ecuación

$$\texttt{unfoldB}\ f\ (\texttt{foldBin}\ (v, f')\ t) = t$$

Para ejemplificar lo anterior consideremos los siguientes árboles:

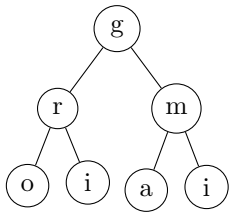


Figura 4.2: `t1`

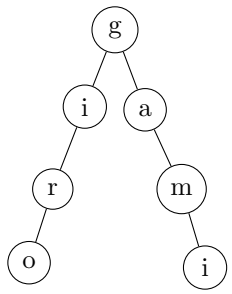


Figura 4.3: `t2`

Que corresponden a las siguientes definiciones en Haskell:

```
t1 :: BinT Char
t1 = (Node 'g'
        (Node 'r'
            (Node 'o' Void Void)
            (Node 'i' Void Void))
        (Node 'm'
            (Node 'a' Void Void)
            (Node 'i' Void Void)))

t2 :: BinT Char
t2 = (Node 'g'
        (Node 'i'
            (Node 'r'
                (Node 'o' Void Void)
                Void)
            Void)
        (Node 'a'
            Void
            (Node 'm'
                Void
                (Node 'i' Void Void))))
```

Al aplicar la función `inroderF` que es una instancia de `foldBin` sobre ambos árboles el resultado es el mismo, la cadena *"origami"*, es decir

```
inorderF t1 = "origami"

inorderF t2 = "origami"
```

Por lo que ambos árboles podrían ser resultado de la llamada `unfoldB f "origami"`.

Capítulo 5

Árboles binarios de búsqueda

Los árboles binarios de búsqueda son un caso particular de los árboles con información en todos los nodos con la característica de que lo árboles de búsqueda están ordenados[2].

Los nodos de un árbol binario de búsqueda cumplen la propiedad de que todos los elementos en subárbol izquierdo son menores que la raíz y todos los elementos almacenados en los nodos del subárbol derecho son mayores o iguales a la raíz [27], es esta la propiedad que genera el orden en los árboles. El almacenamiento en orden que se sigue en los árboles binarios de búsqueda permite realizar las operaciones básicas de una estructura de datos (buscar, agregar y eliminar un elemento) de forma más eficiente en comparación con las listas o los árboles binarios sin orden. Es importante notar que el tipo de los elementos a almacenarse en un árbol binario de búsqueda debe ser un tipo que tenga la propiedad de comparación u orden en sus elementos, esto para poder garantizar un orden entre ellos.

Una representación gráfica de un árbol binario de búsqueda se puede ver en la figura 5.1.

En este ejemplo los elementos del árbol son de tipo `Int` el cual tiene un orden definido y se puede observar que los nodos de este árbol cumplen la propiedad antes mencionada.

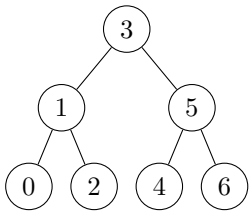


Figura 5.1: Representación gráfica de un árbol binario de búsqueda

5.1. Especificación Formal

Cabe aclarar que a pesar de tratarse de estructuras de datos distintas, los árboles definidos en el capítulo anterior y los árboles binarios de búsqueda son estructuralmente iguales, es decir, que sus constructores se comportan de la misma manera. Esto no es necesariamente cierto en todos los lenguajes de programación, en Java por ejemplo desde la definición de la estructura se tiene que garantizar que los elementos sean de un conjunto con un orden definido. Sin embargo para la especificación formal y mas aún para la definición del tipo de dato algebraico en Haskell se trata esencialmente de la misma estructura cambiando únicamente los nombres de los constructores y es hasta la definición de las funciones definidas sobre esta estructura que se notará una diferencia respecto a los árboles binarios con información en todos los nodos.

Definición 5.1 (Definición formal de árboles binarios de búsqueda [28][29])**.** *Un árbol binario de búsqueda de elementos de un tipo A sobre el cual hay un orden total definido, es un árbol binario que satisface las siguientes condiciones respecto al elemento almacenado en la raíz x:*

1. *Para cada elemento u almacenado en un nodo del subárbol izquierdo $u < x$.*
2. *Para cada elemento v almacenado en un nodo del subárbol derecho $v \geqslant x$*

Ahora se definen los árboles binarios de búsqueda de forma inductiva, para poder definirlo como un tipo de dato en Haskell.

Definición 5.2 (Definición inductiva de árboles binarios de búsqueda)**.** *Un árbol binario de búsqueda de elementos de un tipo A se define recursivamente como:*

- `Void` *es un árbol binario de búsqueda*
- *Si x es un elemento de tipo A, y t_1 y t_2 son árboles binarios de búsqueda, entonces (*`Node` *x t_1 t_2) es un árbol binario de búsqueda.*
- *Son todos*

La Definición 5.1 no puede incluirse en un tipo de dato algebraico en Haskell, por lo que se define el tipo que abstrae solo la construcción estructural de los árboles binarios de búsqueda.

Definición 5.3 (Tipo de dato `BSrchT`)**.**

```
data BSrchT a = Void | Node a (BSrchT a) (BSrchT a)
```

La Definición 5.2 genera el siguiente principio de inducción

Definición 5.4 (Principio de Inducción Estructural para BSrchT)**.** *Sea $\mathcal{P}$ una propiedad acerca de árboles de elementos de A. Para probar que para todo árbol $t \in$* `BSrchT` *cumple la propiedad $\mathcal{P}$, basta demostrar lo siguiente:*

- *Base: el constructor* `Void` *cumple $\mathcal{P}$.*
- *Hipótesis de Inducción: suponer que $\mathcal{P}$ se cumple para los árboles t_1 y t_2.*
- *Paso Inductivo: mostrar usando la hipótesis de Inducción que (*`Node` *x t_1 t_2) cumplen $\mathcal{P}$ con $x \in A$.*

A continuación se define el álgebra semántica correspondiente al tipo de dato 5.3.

Definición 5.5 (Álgebra Semántica de `BSrchT`)**.** *Definición de álgebra*

```
type SrchAlgebra a t = (t, a-> t -> t -> t)
```

Por último definamos el operador fold para `BSrchT`

Definición 5.6 (Fold General)**.** *Se define la función* `foldSrch` *como sigue:*

```
foldSrch :: SrchAlgebra a t -> BSrchT a -> t
foldSrch (b, f) = fold where
    fold Void         = b
    fold (Node x l r) = f x (fold l) (fold r)
```

Es fácil observar que las definiciones anteriores son, en términos prácticos, iguales a las de la estructura `BinT` salvo por los identificadores tanto de las funciones como de los tipos de dato.

5.2. Propiedades del fold

Como vimos en la sección anterior, las definiciones de `BSrchT` y `BinT` son iguales salvo por los nombres de los constructores, por lo que la Propiedad Universal y el Principio de Fusión del operador `foldSrch` se heredan directamente

de estas mismas propiedades para el operador `foldBin`. Tratarlo como un operador diferente resultaría en repetir las demostraciones que ya se presentaron en el capítulo anterior. Es por esto que en esta sección nos limitaremos a enunciar los principios sobre el nuevo operador `foldSrch`.

5.2.1. Propiedad Universal

Proposición 5.1 (Propiedad Universal de fold para árboles binarios de búsqueda)**.** *La Propiedad Universal se puede enunciar con la siguiente equivalencia entre las definiciones de la función g.*

$$\begin{aligned} &g\ \texttt{Void} = v \\ &g\ (\texttt{Node}\ x\ t_1\ t_1) = n\ x\ (g\ t_1)\ (g\ t_2) \end{aligned} \quad \Longleftrightarrow \quad g = \texttt{foldSrch}\ (v, n)$$

5.2.2. Principio de Fusión

Lema 5.1 (Principio de Fusión para árboles binarios de búsqueda)**.** *Sea h una función tal que cumple las siguientes condiciones*

- $h\ w = v$
- $h\ (g\ x\ y_1\ y_2) = n\ x\ (h\ y_1)\ (h\ y_2)$

con n una función estricta, entonces es cierto que

$$h \cdot \texttt{foldSrch}\ (w, g) = \texttt{foldSrch}\ (v, n)$$

5.3. Construcción

Como vimos anteriormente no hay forma de garantizar que una instancia de `BSrchT` cumpla las propiedades de un árbol binario de búsqueda de la definición 5.1, por ejemplo consideremos el siguiente árbol `t`.

```
t :: BSrchT Int
t = (Node 25
          (Node 94 Void Void)
          (Node 6 Void Void))
```

que gráficamente se ve como se muestra en la figura 5.2

El árbol `t` no se trata de un árbol binario de búsqueda ya que no cumple la propiedad de estar ordenado, pues el elemento en el subárbol izquierdo de la raíz es mayor y el elemento en el subárbol derecho es menor. Sin embargo, sí se trata

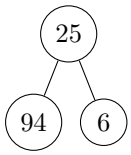


Figura 5.2: `t`

de un árbol del tipo `BSrchT` pues está bien construido con los constructores de este tipo.

Es por esto que surge la necesidad de definir una función constructora de `BSrchT` que garantice que el árbol resultante es un árbol binario de búsqueda, esta función debe recibir una colección de elementos con un orden definido, que representaremos como una lista en la implementación, y regresar un `BSrchT` ordenado. Para implementar esta función se define primero una función `add` que agrega un elemento a un `BSrchT` como sigue:

```
add :: Ord a => a -> BSrchT a -> BSrchT a
add e Void          = (Node e Void Void)
add e (Node x l r)
 | e < x            = Node x l' r
 | otherwise        = Node x l r'
 where l' = add e l
       r' = add e r
```

La función `add` supone que el árbol de tipo `BSrchT` que recibe como argumento está correctamente construido, es decir esta ordenado y preserva este orden cumpliendo las propiedades de los árboles binarios de búsqueda al agregar el nuevo elemento en el subárbol izquierdo si este fuera menor a la raíz y en el subárbol derecho en otro caso.

Utilizando la función anterior se puede definir `construct` de la siguiente forma:

```
construct :: Ord a => [a] -> BSrchT a
construct []     = Void
construct (x:xs) = add x (construct xs)
```

Esta función va agregando uno a uno los elementos de la lista al árbol binario de búsqueda, para agregarlos hace uso de la función `add` que garantiza la preservación de la propiedad de orden en el árbol, por lo que se tiene la garantía de que el árbol generado a partir de la función `construct` es un árbol binario de búsqueda.

Una observación interesante sobre la función `construct` es que sigue el patrón de reescritura del operador `fold` para listas definido en el capítulo 2 por lo que puede definirse con el estilo origami.

```
construct :: Ord a => [a] -> BSrchT a
construct = foldr add Void
```

5.3.1. Constructor inteligente

Además de la función que construye un `BSrchT` a partir de una lista, se puede definir un constructor inteligente. En Haskell los constructores inteligentes son un *idiom*[1] en el que se definen funciones que encapsulan el comportamiento de cada constructor del tipo de dato para añadir restricciones adicionales en la construcción de valores[2]. Esta técnica consiste en definir una función por cada uno de los constructores del tipo de dato conservando el nombre y usar estas funciones en lugar de usar explícitamente el constructor[9].

Por ejemplo, los constructores inteligentes para el tipo `BSrchT` serían los siguientes:

```
void :: BSrchT
void = Void

node :: Ord a => a -> BSrchT a -> BSrchT a -> BSrchT a
node x l r
 | (x >: l) && (x <: r) = Node x l r
 | otherwise            = error "No hay orden"
```

La función `void` es una función que no recibe parámetros y que regresa siempre el árbol `Void`, mientras que la función `node` recibe los mismos argumentos que el constructor `Node` y construye un `BSrchT` solo si cumple las propiedades de árboles binarios de búsqueda, usando los operadores `(>:)` y `(<:)` que verifican si un valor es mayor o igual a todos elementos de un árbol y si un valor es menor a todos los elementos de un árbol respectivamente que definen como sigue:

```
(>:) :: Ord a => a -> SeaBinTree a -> Bool
(>:) _ Void         = True
(>:) e (Node l x r) = e >= x && (e >: l) && (e >: r)

(<:) :: Ord a => a -> SeaBinTree a -> Bool
(<:) _ Void         = True
(<:) e (Node l x r) = e < x && (e <: l) && (e <: r)
```

[1] Un idiom es una convención no escrita que se usa al escribir código en un lenguaje de programación, un ejemplo es usar nombres en mayúsculas para definir atributos finales en Java.

[2] Smart Constructors: `https://wiki.haskell.org/Smart_constructors`

Observemos que estos operadores están definidos siguiendo el patrón de reescritura que encapsula el operador `foldSrch` por lo que podemos reescribirlos como instancias de él de la siguiente forma:

```
(>:) :: Ord a => a -> SeaBinTree a -> Bool
(>:) e = foldSrch (True, f)
 where f x l r = x >= e && l && r

(<:) :: Ord a => a -> SeaBinTree a -> Bool
(<:) e = foldSrch (True, f)
 where f x l r = x < e && l && r
```

Y de esta forma se utilizan las funciones `void` y `node` en lugar de los constructores `Void` y `Node` para generar árboles binarios de búsqueda. Por ejemplo el árbol

Figura 5.3: `t1`

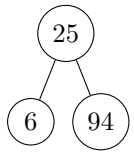


Figura 5.4: `t`

Se define con los constructores inteligentes de la siguiente manera:

```
t1 :: BSrchT Int
t1 = node 25 (node 6 void void) (node 94 void void)
```

Y los constructores inteligentes se encargan de verificar que cumpla las propiedades de un árbol binario de búsqueda. Si se usan estos constructores para definir el árbol en la figura `t` 5.2

```
t :: BSrchT Int
t = node 25 (node 94 void void) (node 6 void void)
```

Se lanzaría el error del constructor inteligente `node` que dice que no es posible construir el árbol porque no está ordenado.

5.4. Ejemplos

En esta sección se presentan ejemplos de funciones reescritas con el operador `foldSrch` así como algunas funciones que no pueden reescribirse con el estilo de programación origami.

5.4.1. Funciones con patrón de reescritura con fold

root Regresa la raíz del árbol

```
root :: Ord a => BSrchT a -> a
root Void          = error "No hay elementos"
root (Node x _ _) = x

rootF :: Ord a => BSrchT a -> a
rootF = foldSrch (error "No hay elementos",
                  (\x _ _ -> x))
```

Resulta interesante analizar este ejemplo en particular pues el caso base definido para el constructor `Void` es un error, lo cual podría creerse que resultaría en un error de tipos, sin embargo en Haskell el tipo de un error es dado por una variable de tipo que toma su valor a conveniencia del programa, en este ejemplo `error "No hay elementos":: a`, que se lee como `error "No hay elementos"` tiene tipo `a`.

aplana Regresa una lista ordenada con los elementos del árbol

```
aplana :: Ord a => BSrchT a -> [a]
aplana Void.         = []
aplana (Node x l r) = (aplana l) ++ x:(aplana r)

aplanaF :: Ord a => BSrchT a -> [a]
aplanaF = foldSrch ([], (\x l r -> l ++ x:r))
```

mapW recibe una función y la aplica a todos los elementos del árbol

```
mapW ::Ord a => (a -> b) -> BSrchT a -> BSrchT b
mapW _ Void          = Void
mapW f (Node x l r) = Node (f x)
                           (mapW f l)
                           (mapW f r)

mapWF ::Ord a => (a -> b) -> BSrchT a -> BSrchT b
mapWF f = foldSrch (Void,
                    (\x l r -> Node (f x) l r))
```

Observese que se trata de un `map` débil, pues al no restringir la función que puede recibir como parámetro no hay garantía de que el resultado siga siendo un árbol binario de búsqueda. Este es un ejemplo de que a pesar de tratarse de la misma definición de los árboles del capítulo anterior se debe verificar las propiedades de los árboles binarios de búsqueda cada vez que se aplica una función y devuelve como resultado un árbol para garantizar que este sea de búsqueda. En Haskell no es posible hacer esto en la misma

definición, sin embargo existen herramientas de verificación compatibles con el lenguaje como lo son LiquidHaskell[30][31], Agda[32], Coq[33], entre algunas otras. Otra opción es realizar esta verificación a mano pero en programas o árboles muy grandes esta opción se vuelve mas compleja.

5.4.2. Funciones que no se pueden reescribir con fold

Para ejemplificar las funciones que no se pueden reescribir como instancias de `foldSrch` definiremos las funciones `maxT` y `minT` para las que es necesario definir un test de vacío es decir una función que diga si un árbol es vacío o no, definida como sigue:

```
isvoid ::  BSrchT a -> Bool
isvoid Void = True
isvoid _ = False
```

maxT Regresa el elemento mas grande del árbol

```
maxT :: Ord a => BSrchT a -> a
maxT Void            = error "No hay elementos"
maxT (Node x l r)    = f x (maxT l) (maxT r)
 where f a b c = if isvoid r then a else c
```

Esta función a simple vista podría parecer que está definida siguiendo el patrón de reescritura del operador `foldSrch`, un caso base para el constructor `Void` y una función que toma como argumentos la raíz y las llamadas recursivas de los subárboles en el caso del constructor `Node` (para este ejemplo la función `f`), sin embargo, no puede reescribirse como una instancia de `foldSrch` pues en la función `f`, para evaluar la guardia del `if`, se usa explícitamente el subárbol `r` para verificar si se trata del árbol vacío y con el operador `fold` se pierde la información de la construcción del árbol pues es reemplazada por el resultado de la llamada recursiva sobre él.

minT Regresa el elemento mas pequeño del árbol

```
minT:: Ord a => BSrchT a -> a
minT Void          = error "No hay elementos"
minT (Node x l r) = f x (minT l) (minT r)
 where f a b c = if isvoid l then a else b
```

Al igual que en la función `maxT`, `minT` no se puede reescribir usando `foldSrch` pues es necesario obtener más información de la construcción del subárbol izquierdo para poder obtener el resultado.

delete Elimina un elemento del árbol.

```
delete :: Ord a => a -> BSrchT a -> BSrchT a
delete _ Void         = Void
delete e (Node x l r)
 | er && (l == Void)  = r
 | er && (r == Void)  = l
 | er                 = (Node m dl r)
 | e < x = Node x (delete e l) r
 | otherwise = Node l x (delete e r)
 where er  = x == e
       m = maxT l
       dl  = delete m l
```

Aunque pareciera que la función `delete` si sigue el patrón de reescritura de `foldSrch` no es así pues en el caso recursivo se requiere más información de la construcción del árbol al igual que en los ejemplos anteriores.

5.4.3. Ejemplos Avanzados

busca busca un elemento en el árbol.

```
busca :: Ord a => BSrchT a -> Bool
busca Void            = False
 busca e (Node x l r)
  | e == x            = True
  | e < x             = (busca e l)
  | otherwise         = (busca e r)

buscaF :: Ord a => a -> BSrchT a -> Bool
buscaF a = foldSrch (False,
          (\x l r ->
            if (x == a)
              then True
              else
                if (a < x)
                  then l
                  else r))
```

En este ejemplo se puede observar que el desempeño de ambas funciones es el mismo gracias a la naturaleza "perezosa" de Haskell en donde los casos `then` y `else` de `if` no se evaluarán a menos que la guardia indique que es necesario.

5.5. Unfold

Ahora se define el operador `unfoldS` que es una especie de operador *inverso* para `foldSrch`.

Definición 5.7 (Unfold de árboles binarios de búsqueda)**.** *Definido en* Haskell *como sigue:*

```
unfoldS :: (b -> Maybe(a,b,b)) -> b -> BSrchT a
unfoldS f b = case (f b) of
  Just(x,l,r) ->
   Node x (unfoldS f l) (unfoldS f r)
  Nothing     -> Void
```

Al igual que los operadores `unfold` vistos hasta ahora, este no es necesariamente determinista en el sentido de que la ecuación

$$\texttt{unfoldS}\ f\ (\texttt{foldSrch}\ (v, f')\ t) = t$$

sea cierta en todos los casos y va a depender de las funciones f y f' con las que se dobla y desdobla el árbol t. Para esta estructura hay que ser especialmente cuidadoso al definir la función f' para garantizar que el resultado de la función `unfoldS` cumpla las propiedades de los árboles binarios de búsqueda. Como ejemplo consideremos los siguientes árboles binarios de búsqueda:

Figura 5.5: `t1`

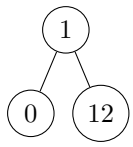
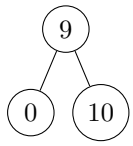


Figura 5.6: `t2`



Definidos como instancias de `BinT` como sigue:

```
t1 :: BSrchT Int
t1 = (Node 1 (Node 0 Void Void) (Node 12 Void Void))

t1 :: BSrchT Int
t2 = (Node 9 (Node 0 Void Void) (Node 10 Void Void))
```

Consideremos también una función `f` que calcula la suma de los cubos de sus parámetros definida de la siguiente forma:

```
f :: Int -> Int -> Int -> Int
f x y z = x ^ 3 + y ^ 3 + z ^ 3
```

Las llamadas a la función `foldSrch` con el parámetro `f` y cada uno de los árboles definidos anteriormente genera el mismo resultado. Es decir

```
foldBin (0,f) t1 = 1729

foldBin (0,f) t2 = 1729
```

Por lo que ambos árboles podrían ser el resultado de `unfoldB f' 1729` dependiendo de la definición que se dé para `f'`.

5.6. Tipos Refinados

Una alternativa para definir árboles binarios de búsqueda, en donde se garantice la propiedad de orden en el almacenamiento, es el uso de tipos refinados o de refinamiento. Los tipos refinados son tipos sobre los cuales se agregan propiedades críticas que deben de cumplir mediante predicados lógicos [34].

El sistema de tipos de Haskell no cuenta con tipos refinados, sin embargo existe el lenguaje LiquidHaskell en el que se extiende el sistema de tipos de Haskell para permitir un sistema de tipos más expresivo al usar tipos refinados. Este lenguaje conserva la sintaxis y semántica de Haskell y puede verse (en términos muy generales) como una extensión de él.

La definición de árboles binarios de búsqueda en LiquidHaskell es la siguiente:

```
{-@ type BSTL a X = BST { v:a | v < X }  @-}
{-@ type BSTR a X = BST { v:a | X <= v }  @-}

{-@ data BST a =
        Void
      | Node {root :: a,
              left :: BSTL a root,
```

```
                right :: BSTR a root}
@-}
```

Los tipos refinados se escriben entre llaves y el símbolo `-@`, ejemplo: {`-@ ... @-`}. Primero se definen los tipos `BSTL` que corresponde al tipo del subárbol izquierdo y es aquí en donde se realiza la verificación de que los elementos de este árbol son menores a la raíz y el tipo`BSTR` que corresponden al subárbol derecho y se verifica que los elementos sean mayores o iguales a la raíz. Con estos tipos de define `BST` con los constructores `Void` para el árbol vacío y `Node` en donde la raíz tiene el tipo de los elementos a almacenar, el subárbol derecho tiene tipo `BSTL` y el subárbol izquierdo tiene tipo `BSTR`.

LiquidHaskell es al igual que Haskell un lenguaje de programación funcional, por lo que también se puede programar usando el método de programación origami y todas las definiciones y propiedades que hemos visto hasta ahora se preservan en este lenguaje.

Definir árboles binarios de búsqueda de esta forma es una alternativa interesante y útil para garantizar el comportamiento de estos, pero profundizar en ellos se encuentra fuera del alcance de este trabajo por lo que sólo se presenta como un ejemplo.

Capítulo 6

Árboles Cartesianos

Un árbol cartesiano es un árbol binario construido a partir de una secuencia de números de tal forma que se preserva el orden que tenía la secuencia original, es decir, que para cada nodo todos los elementos de su subárbol izquierdo aparecían antes que el elemento del nodo en la secuencia original y de la misma forma, todos los elementos del subrárbol derecho aparecían después que el valor del nodo en la secuencia original. De tal forma que al hacer un recorrido *in-order* del árbol se obtiene la secuencia original.

Además de estas características un árbol cartesiano tiene la propiedad de ser un *minHeap*, es decir que para todos los subárboles del árbol se cumple que el valor de la raíz es más pequeño que el valor de sus hijos (también podría ser un *maxHeap* dependiendo de la implementación para este trabajo se toma la propiedad de minHeap)[35].

La representación gráfica de la secuencia `[9,3,7,1,8,12,10,20,15,18,5]` se puede ver en la figura 6.1:

6.1. Especificación formal

Al igual que los árboles de búsqueda, los árboles Cartesianos son un caso particular y estructuralmente equivalente de los árboles binarios con información en todos los nodos. Así que en esta sección daremos las definiciones de la estructura sin profundizar en cada una, pues ya se hizo en capítulos anteriores.

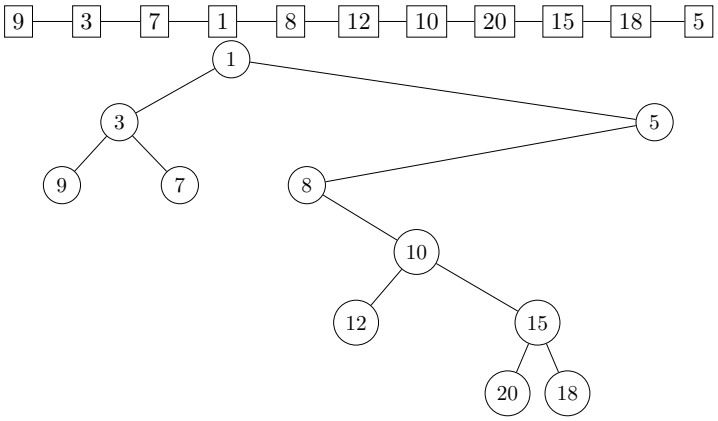


Figura 6.1: Representación gráfica de un árbol Catesiano

Definición 6.1 (Definición formal de árboles cartesianos [35])**.** *Un árbol cartesiano es un árbol binario que cumple las siguientes propiedades:*

1. *Sea x el elemento de un nodo n, para todo elemento y almacenado en un nodo de nivel superior a n, $x < y$, es decir es un minHeap.*
2. *Un recorrido in order de los nodos produce la secuencia a partir de la cual se creó el árbol.*

La definición anterior puede adaptarse para implementaciones en las que los árboles cartesianos sean *maxHeap*, cambiando la primera propiedad en donde la relación entre el x y y es $x > y$.

La segunda propiedad puede escribirse como la siguiente ecuación en Haskell, suponiendo que el árbol es construido a partir de una lista `xs`:

```
inorder (construct xs) = xs
```

En donde `construct` es una función que construye un árbol cartesiano a partir de la lista dada como argumento, esta función es definida más adelante en este capítulo.

Definición 6.2 (Definición inductiva de árboles cartesianos)**.** *Un árbol cartesiano de elementos de un conjunto A se define recursivamente como:*

- `Void` *es un árbol cartesiano*
- *Si $x \in A$, y t_1 y t_2 son árboles cartesianos, entonces (*`Node` *x t_1 t_2) es un árbol cartesiano.*
- *Son todos*

Definido en Haskell como el siguiente tipo de dato algebraico.

Definición 6.3 (Tipo de dato `CartT`)**.**

```
data CartT a = Void | Node a (CartT a) (CartT a)
```

Esta definición genera el siguiente principio de inducción.

Definición 6.4 (Principio de Inducción Estructural para CartT)**.** *Sea $\mathcal{P}$ una propiedad acerca de árboles de elementos de A. Para probar que para todo árbol cartesiano t cumple la propiedad $\mathcal{P}$, basta demostrar lo siguiente:*

- *Base: el constructor* `Void` *cumple $\mathcal{P}$.*
- *Hipótesis de inducción: suponer que $\mathcal{P}$ se cumple para los árboles t_1 y t_2.*
- *Paso Inductivo: mostrar usando la hipótesis de Inducción que (*`Node` *x t_1 t_2) cumplen $\mathcal{P}$ con $x \in A$.*

Definimos ahora el álgebra semántica correspondiente a `CartT`

Definición 6.5 (Álgebra semántica de `CartT`)**.** *Definición de álgebra semántica.*

```
type CartAlgebra a t = (t, a-> t -> t -> t)
```

Por último definamos el operador fold para `CartT`

Definición 6.6 (Fold General)**.** *Se define la función* `foldCart` *como sigue:*

```
foldCart :: CartAlgebra a t -> CartT a -> t
foldCart (b, f) = fold where
    fold Void         = b
    fold (Node x l r) = f x (fold l) (fold r)
```

6.2. Propiedades del fold

El operador `foldCart` hereda las propiedades ya vistas para el operador `foldBin` pues esencialmente se trata de la misma definición.

6.2.1. Propiedad Universal

Proposición 6.1 (Propiedad Universal de fold para árboles Cartesianos)**.** *La Propiedad Universal se puede enunciar con la siguiente equivalencia entre las definiciones de la función g.*

$$\begin{array}{l} g\ \texttt{Void} = v \\ g\ (\texttt{Node}\ x\ t_1\ t_1) = n\ x\ (g\ t_1)\ (g\ t_2) \end{array} \iff g = \texttt{foldCart}\ (v, n)$$

6.2.2. Principio de Fusión

Lema 6.1 (Principio de Fusión para árboles Cartesianos)**.** *Sea h una función tal que cumple las siguientes condiciones*

- $h\ w = v$
- $h\ (g\ x\ y_1\ y_2) = n\ x\ (h\ y_1)\ (h\ y_2)$

con n una función estricta, entonces es cierto que

$$h \cdot \texttt{foldCart}\ (w, g) = \texttt{foldCart}\ (v, n)$$

6.3. Construcción

Como se describió al inicio de este capítulo los árboles Cartesianos son construidos a partir de una secuencia, de tal forma que el árbol resultante es *minheap* y al hacer un recorrido *in-order* del árbol se obtiene la secuencia original.

Existen diferentes algoritmos para la creación de un Árbol Cartesiano, para este trabajo usaremos un algoritmo de construcción basado en la filosofía *divide y vencerás*. Se seleccionó este método pues se trata de un algoritmo recursivo y de naturaleza funcional lo que resulta conveniente para escribirlo en el lenguaje de programación Haskell.

Para nuestra implementación supondremos que la secuencia con la cual se va a construir el árbol Cartesiano se encuentra almacenada en una lista, y de esta forma para generar un `CartT` se siguen los siguientes pasos:

1. Se encuentra el elemento más pequeño de la lista (min).
2. Se divide la lista en los elementos que aparecen antes del mas pequeño (l_1) y aquellos que aparecen después (l_2).
3. Se crea un nuevo árbol Cartesiano que tiene como raíz a min como subárbol izquierdo el generado recursivamente a partir de l_1 y como subárbol derecho el resultante de la llamada recursiva con l_2.

6.3.1. Implementación

Para la implementación en Haskell del algoritmo anterior primero se definen un par de funciones auxiliares que son las encargadas de dividir la lista, para poder así hacer las llamadas recursivas.

Primero se define la función `subto` que recibe un elemento, una lista y regresa la sublista hasta encontrar dicho elemento.

```
subto :: (Ord a) => [a] -> a -> [a]
subto [] _      = []
subto (x:xs) e
 | x == e       = []
 | otherwise    = x:(subto xs e)
```

Ahora definimos `subfrom` que recibe un elemento y regresa la sublista a partir de este elemento.

```
subfrom :: (Ord a) => [a] -> a -> [a]
subfrom [] _      = []
subfrom (x:xs) e
 | x == e         = xs
 | otherwise      = subfrom xs e
```

Con estas funciones se puede definir el constructor para árboles Cartesianos a partir de una lista, definido con la función `construct`.

Definición 6.7 (Constructor de árboles Cartesianos)**.** *Definido en* Haskell *como sigue:*

```
construct :: (Ord a) => [a] -> CartT a
construct []  = Void
construct lst = Node root left right where
 root  = minimum lst
 left  = construct (subto lst root)
 right = construct (subfrom lst root)
```

Por último definamos la función `tolist` que transforma el árbol en la lista original

```
tolist :: (Ord a) => CartT a -> [a]
tolist Void         = []
tolist (Node x l r) = (tolist l) ++ x:(tolist r)
```

6.3.2. Implementación Origami

Ahora se dará una implementación del algoritmo de construcción de árboles Cartesianos con programación origami.

Para este propósito primero se reescribirán las funciones `subto` y `subfrom` a su versión origami, es decir haciendo uso del operador `fold`. Como la recursión en estas funciones se hace sobre listas, el operador que se utiliza es `foldList` 2.5 resultando las siguientes funciones.

```
subtoF :: (Ord a) => a -> [a] -> [a]
subtoF e = foldList ([],
            (\x y -> if x == e then [] else (x:y)))

subfromF :: (Ord a) => a -> [a] -> [a]
subfromF e = foldList ([],
            (\x y -> if x == e then y else (e:y)))
```

Ahora observemos como la función `construct` 6.7 no sigue tal cual el patrón encapsulado por el operador `fold`, por lo que no puede reescribirse usando `foldList`. Esto se debe a que la recursión no va disminuyendo la entrada quitando la cabeza de la lista como lo hacen las funciones anteriores y es este el patrón definido con `fold`, lo que se conoce como recursión lineal. Simplemente la reescribimos haciendo uso de las funciones definidas en programación origami `subtoF` y `subfromF`:

Definición 6.8 (Nuevo constructor de árboles cartesianos)**.** *Definido en* Haskell *como sigue:*

```
constructF :: (Ord a) => [a] -> CartT a
constructF []  = Void
constructF lst = Node root left right where
 root  = minimum lst
 left  = constructF (subtoF lst root)
 right = constructF (subfromF lst root)
```

Ahora se reescribe la función `tolist` a programación origami, en este caso sí se utiliza el operador `foldCart` pues la recursión es sobre árboles cartesianos.

```
tolistF :: (Ord a) => CartT a -> [a]
tolsitF = foldCart([],(\x l r -> l ++ x:r))
```

Esta construcción de árboles cartesianos es un ejemplo del uso de programación origami, es equivalente a la construcción dada en la sección anterior pero haciendo uso de otro estilo de programación. Se puede observar que las funciones definidas en origami son más compactas aunque se pierde un poco la legibilidad en comparación con las funciones escritas con recursión estructural.

Capítulo 7

Rosadelfas

Las rosadelfas son estructuras arbóreas que, a diferencia de las vistas hasta ahora, no limitan el número de hijos que un nodo puede tener. Para lograrlo cada nodo almacena una lista que contienen a todos sus hijos[21][36]. Gráficamente las rosadelfas se pueden ver de la forma que se muestra en la figura 7.1.

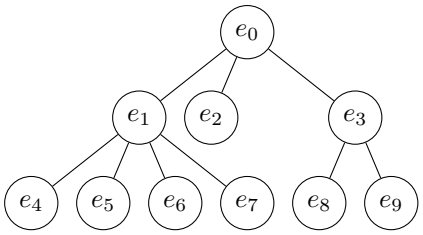


Figura 7.1: Representación gráfica de una rosadelfa

7.1. Especificación formal

La definición formal de una rosadelfa es

Definición 7.1 (Definición inductiva de rosadelfa[21])**.** *Una Rosadelfa de elementos del conjunto A se define como sigue:*

- `Void` *es una rosadelfa*
- *Si* $x \in A$ *y lst es una lista de rosadelfas. Entonces (*`Node` *x lst) es una rosadelfa.*
- *Son todas*

Hay que observar que según la definición 7.1 `Void` es la forma de construir una rosadelfa vacía, y en el constructor `Node`, el número de hijos de un nodo será la longitud de la lista *lst*, siendo hojas los nodos en donde *lst* es vacía.

Implementar la definición 7.1 en Haskell resulta en el siguiente tipo de dato algebraico:

Definición 7.2 (Tipo de dato `Rosadelfa`)**.**

```
data Rosadelfa a = Void | Node a [Rosadelfa a]
```

La definición inductiva de rosadelfa (7.1) genera el principio de inducción enunciado a continuación.

Definición 7.3 (Principio de Inducción Estructural para Rosadelfa)**.** *Sea $\mathcal{P}$ una propiedad acerca de Rosadelfas. Para probar que toda Rosadelfa cumple la propiedad $\mathcal{P}$, basta demostrar lo siguiente:*

- *Base: el constructor* `Void` *cumple $\mathcal{P}$.*
- *Hipótesis de inducción: suponer que $\mathcal{P}$ es cierta para los árboles t_1, t_2, ..., t_n.*
- *Paso Inductivo: mostrar usando la hipótesis de Inducción que (*`Node` *x [t_1,t_2 ,..., t_n]) cumplen $\mathcal{P}$ con $x \in A$.*

Se define ahora el álgebra semántica asociada a la estructura con el siguiente sinónimo en Haskell:

Definición 7.4 (Álgebra semántica de rosadelfa)**.** *Definición de álgebra semántica de rosadelfa.*

```
type AlgebraRosa a r = (r, a -> [r] -> r)
```

El patrón general que se busca encapsular con el operador `fold` de la estructura es el definido con la función h

```
h Void          = v
h (Node x lst) = n x (map h lst)
```

Como la construcción de una rosadelfa depende de una lista es necesario hacer uso de la función `map` de listas para aplicar recursivamente la función `h` a cada uno de los hijos del nodo.

Capturando el patrón, el operador `foldRosa` se define como sigue:

```
foldRosa :: AlgebraRosa a t -> Rosadelfa a -> t
foldRosa (void, node) = fold where
   fold Void = void
   fold (Node e lst) = node e (map (fold) lst)
```

Esta definición puede hacerse mas "pura" si utilizamos la definición Origami de `map` definida en el capítulo 2 resultando la siguiente función.

Definición 7.5 (Fold General de Rosadelfa)**.**

```
foldRosa :: AlgebraRosa a t -> Rosadelfa a -> t
foldRosa (b, f) = fold where
   fold Void         = b
   fold (Node e lst) = f e (mapF (fold) lst)
```

7.2. Propiedades del operador foldRosa

En esta sección se estudiará cómo se ajustan la Propiedad Universal y el Principio de Fusión en la nueva estructura definida en este capítulo.

7.2.1. Propiedad Universal

Probemos la unicidad del operador `foldRosa`

Proposición 7.1 (Propiedad Universal de `foldRosa`)**.** *Para cualquier función* `g` *se cumple que:*

$$\begin{array}{l} g\ \texttt{Void} = v \\ g\ (\texttt{Node}\ x\ ts) = f\ x\ (\texttt{map}\ g\ ts) \end{array} \quad \Longleftrightarrow \quad g = \texttt{foldRosa}\ (v, f)$$

Demostración. ($\Rightarrow$ Utilizando el principio de inducción para rosadelfa 7.3)

Base

$$\begin{array}{cc} & g\ \texttt{Void} \\ = & \{\text{Definición de la función } g\} \\ & v \\ = & \{\text{Definición de } \texttt{foldRosa}\} \\ & \texttt{foldRosa}\ (v, f)\ \texttt{Void} \end{array}$$

H.I. Sean t_1, t_2, ... , t_n rosadelfas, entonces $g\ t_i = \texttt{foldRosa}\ (v, f)\ t_1$ para toda $i \in \{1, ..., n\}$

Paso Inductivo

$$\begin{array}{cc} & g\ (\texttt{Node}\ x\ [t_1, t_2, \ldots, t_n]) \\ = & \{\text{Definición de la función } g\} \\ & f\ x\ (\texttt{map}\ g\ [t_1, t_2, \ldots, t_n]) \\ = & \{\text{Hipótesis de Inducción}\} \\ & f\ x\ (\texttt{map}\ (\texttt{foldRosa}\ (v, f))\ [t_1, t_2, \ldots, t_n]) \\ = & \{\text{Definición de } \texttt{foldRosa}\} \\ & \texttt{foldRosa}\ (v, f)\ (\texttt{Node}\ x\ [t_1, t_2, \ldots, t_n]) \end{array}$$

($\Leftarrow$)

$$
\begin{array}{cc}
 & q \\
= & \{\text{Hipótesis}\} \\
 & \texttt{foldRosa}\ (v, f) \\
\Rightarrow & \{\text{Sustituyendo en la defincición de } \texttt{foldRosa}\} \\
 & g\ \texttt{Void} = v \\
 & g\ (\texttt{Node}\ x\ [t_1, t_2, \ldots, t_n]) = f\ x\ (\texttt{map}\ g\ [t_1, t_2, \ldots, t_n])
\end{array}
$$

□

7.2.2. Principio de Fusión

A partir de la siguiente ecuación

$$h \cdot \texttt{foldRosa}\ (w, g) = \texttt{foldRosa}\ (v, f)$$

Aplicando la Propiedad Universal correspondiente se hace un proceso de síntesis.

1. Caso `Void`:
$$(h \cdot \texttt{foldRosa}\ (w, g))\ \texttt{Void} = v$$
2. Caso `Node`:
$$(h \cdot \texttt{foldRosa}\ (w, g))\ (\texttt{Node}\ x\ ts) = f\ x\ (\texttt{map}\ (\texttt{foldRosa}\ (w, g))\ ts)$$

Primero resolvemos la ecuación correpondiente al constructor `Void`

$$
\begin{array}{cc}
 & (h \cdot \texttt{foldRosa}\ (w, g))\ \texttt{Void} = v \\
\iff & \{\text{Definición de Composición}\} \\
 & h\ (\texttt{foldRosa}\ (w, g)\ \texttt{Void}) = v \\
\iff & \{\text{Definición de } \texttt{foldRosa}\} \\
 & h\ w = v
\end{array}
$$

Ahora se realiza el proceso de síntesis sobre la ecuación que corresponde al constructor `Node`

$$
\begin{array}{cc}
 & (h \cdot \texttt{foldRosa}\ (w, g))\ (\texttt{Node}\ x\ ts) = f\ x\ (\texttt{map}\ (\texttt{foldRosa}\ (w, g))\ ts) \\
\iff & \{\text{Definición de Composición}\} \\
 & h\ (\texttt{foldRosa}\ (w, g)\ (\texttt{Node}\ x\ ts)) = f\ x\ (\texttt{map}\ (\texttt{foldRosa}\ (w, g))\ ts) \\
\iff & \{\text{Definición de } \texttt{foldRosa}\} \\
 & h\ (g\ x\ (\texttt{map}\ (\texttt{foldRosa}\ (w, g))\ ts)) = f\ x\ (\texttt{map}\ (\texttt{foldRosa}\ (w, g))\ ts) \\
\Rightarrow & \{\text{Renombramos } ys = \texttt{map}\ (\texttt{foldRosa}\ (w, g))\ ts\} \\
 & h\ (g\ x\ ys) = f\ x\ ys \\
\iff & \{\text{Definición de Composición}\} \\
 & h \cdot g = f
\end{array}
$$

Con estas condiciones se puede definir el principio de fusión para rosadelfas.

Lema 7.1 (Principio de Fusión para `Rosadelfa`)**.** *Sea h una función tal que cumple las siguientes condiciones*

- $h\ w = v$
- $h \cdot g = f$

con f una función estrícta, entonces es cierto que

$$h \cdot \texttt{foldRosa}\ (w, g) = \texttt{foldRosa}\ (v, f)$$

7.3. Ejemplos

Ahora se muestran algunos ejemplos de funciones que pueden ser implementadas como instancias de `foldSrch` así como algunas funciones que no pueden reescribirse con el estilo de programación origami.

7.3.1. Funciones con patrón de reescritura con fold

nodos Calcula el número de nodos de una rosadelfa

```
nodos :: Rosadelfa a -> Int
nodos Void       = 0
nodos (Node e l) = 1 + sum (map nodos l)

nodosF :: Rosadelfa s -> Int
nodosF = foldRosa (0, \_ y-> 1 + sum y)
```

altura Calcula la altura de una rosadelfa

```
altura :: Rosadelfa a -> Int
altura Void         = 0
altura (Node e lst) = 1 + maxlist (map altura lst)
  where maxlist = foldl (max) 0

alturaF :: Rosadelfa a -> Int
alturaF = foldRosa (0,\_ y -> 1 + maxlist y)
  where maxlist = foldl (max) 0
```

mapRosa Aplica una función a cada elemento del árbol

```
mapRosa :: (a -> b) -> Rosadelfa a -> Rosadelfa b
mapRosa _ Void         = Void
mapRosa f (Node x lst) = Node (f x) flst
  where flst = map (\x -> mapRosa f x) lst

mapRosaF :: (a -> b) -> Rosadelfa a -> Rosadelfa b
mapRosaF f = foldRosa (Void,\x l -> Node (f x) l)
```

elemRosa Dice si un elemento es parte de la rosadelfa

```
elemRosa :: (Eq a) => a -> Rosadelfa a -> Bool
elemRosa _ Void = False
elemRosa e (Node x lst)
  | (e == x)  = True
  | otherwise = or (map (\x -> elemRosa e x) lst)

elemRosaF :: (Eq a) => a -> Rosadelfa a -> Bool
elemRosaF e = foldRosa (False,
   \x lst -> if (x == e) then True else or lst)
```

7.3.2. Funciones que no se pueden reescribir con fold

Para ejemplificar las funciones que no se pueden traducir utilizando `foldRosa` se define la función `elimina`. Esta función se encarga de eliminar un elemento de la rosadelfa.

```
elimina ::(Eq a) => a -> Rosadelfa a -> Rosadelfa a
elimina _ Void        = Void
elimina e ros@(Node x [])
  | x == e    = Void
  | otherwise = ros
elimina e (Node x lst)
  | x == e    = Node r (l++t)
  | otherwise = Node x l
  where h = head lst
        r = raiz h
        t = hijos h
        l = map (\x -> elimina e x) lst
```

En este caso la función no puede traducirse pues necesita más información de la construcción de la rosadelfa, como se puede ver en el segundo caso de la función en donde se coincide con aquellas rosadelfas que representan una hoja,

como este caso se trata por separado es imposible reescribirla con el patrón general encapsulado por el operador `foldRosa`.

7.3.3. Ejemplos Avanzados

hojas Regresa el número de hojas que tiene la rosadelfa

```
hojas :: Rosadelfa a -> Int
hojas Void          = 0
hojas (Node x lst)
 | (null lst)       = 1
 | otherwise        = foldl (+) 0 (map hojas lst)

hojasF :: Rosadelfa a -> Int
hojasF = foldRosa  (0,
    \x lst -> if (null lst) then 1 else foldl (+)
        0 lst)
```

recorre Regresa una lista con los elementos de la rosadelfa

```
recorre :: Rosadelfa a -> [a]
recorre Void         = []
recorre (Node x lst) = x:(foldl (++) [] (map
   recorre lst))

recorreF :: Rosadelfa a -> [a]
recorreF = foldRosa ([],\x lst -> x:(foldl (++)
   [] lst))
```

filterR La función filter adaptada para rosadelfas.

```
filterR::(a->Bool) -> Rosadelfa a -> Rosadelfa a
filterR Void = Void
filterR f (Node x lst)
 | f x       = new
 | otherwise = elimina x new
 where new = Node x (map (\y -> filterR f y) lst)

filterRF::(a->Bool) -> Rosadelfa a -> Rosadelfa a
filterRF f = foldRosa (Void,
     \x lst ->
        let new = Node x lst in
            if (f x) then new
                     else (elimina x new))
```

7.4. Unfold

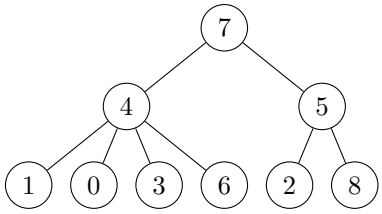
Se define ahora el operador `unfold` para esta estructura con la función `unfoldRosa`.

Definición 7.6 (Unfold de rosadelfas)**.** *Se define en* Haskell *la función* `unfoldRosa` *como sigue:*

```
unfoldRosa::(b -> Maybe(a,[b])) -> b -> Rosadelfa a
unfoldRosa f b = case (f b) of
 Nothing -> Void
 Just (x,lst) -> Node x $ map (unfoldRosa).f lst
```

Para este caso se muestra un ejemplo en el que hay una única solución de la llamada al operador `unfoldRosa`. Consideremos la siguiente `Rosadelfa`:

Figura 7.2: `r`

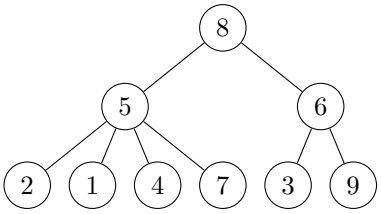


Que se define en Haskell como:

```
r :: Rosadelfa Int
r = (Node 7 [
      (Node 4 [
         (Node 1 []),
         (Node 0 []),
         (Node 3 []),
         (Node 6 [])]),
      (Node 5 [
         (Node 2 []),
         (Node 8 [])])])
```

Si se llama a la función `mapRosaF`, que es una instancia de `foldRosa`, con el argumento `succ` sobre `r`, es decir, `mapRosaF succ r = r'`, en donde `r'` es la siguiente rosadelfa:

Figura 7.3: `r'`



en Haskell:

```
r' :: Rosadelfa Int
r' = (Node 8 [
      (Node 5 [
         (Node 2 []),
         (Node 1 []),
         (Node 4 []),
         (Node 7 [])]),
      (Node 6 [
         (Node 3 []),
         (Node 9 [])])])
```

Se aplica el operador `unfold` con la función `pred'` definida de la siguiente forma:

```
pred' :: (Enum a) => a -> Maybe a
pred' = Just . pred
```

Esta función sólo envuelve el resultado de `pred` en un tipo `Maybe` para que los tipos correspondan con los del operador `unfoldRosa`. Entonces el resultado de la llamada `unfoldRosa pred' r'` es la rosadelfa original `r`, es decir cumple con la ecuación:

$$\texttt{unfoldRosa}\ f'\ (\texttt{foldRosa}\ (v, f)\ r) = r$$

Es importante aclarar que a pesar de que en este caso específico la ecuación si se cumplió, en general esto no va a suceder. Podemos tomar como contraejemplos la generalización de los árboles construidos en la misma sección de capítulos anteriores. Pues recordemos que los árboles binarios son un caso particular de las rosadelfas en donde cada nodo tiene a lo mas dos hijos, y de está forma los árboles definidos en los ejemplos de los capítulos anteriores pueden definirse también como rosadelfas.

En este ejemplo fue sencillo abstraer la función f' por tratarse de funciones sencillas y bien conocidas que operan con el tipo `Int` pero en general encontrar esta función es una tarea que requiere de imaginación y paciencia.

Capítulo 8

Estructuras arbóreas avanzadas

En este capítulo se mostrarán otras estructuras de datos con construcción de árbol. Estas estructuras tienen características y usos específicos dentro de las Ciencias de la Computación. Se estudiarán sólo la definición de la estructura, tanto formal como en Haskell, así como el operador `fold` correspondiente a cada una.

8.1. Árboles Rojinegros

Los árboles Rojinegros son modelados como una mejora sobre los árboles binarios de búsqueda, estructuralmente son casi iguales, salvo por el hecho de que los nodos tienen un atributo de color que sirve para mantener el árbol balanceado, es decir que la diferencia entre el número de nodos del subárbol derecho y el izquierdo no sea muy grande[29].

Para que un árbol binario de búsqueda sea Rojinegro tienen que cumplir las siguientes propiedades:

- Todos los nodos tienen color, rojo o negro.
- La raíz del árbol es negra.
- Todos los `Void` son negros.
- Los hijos de un nodo rojo son siempre negros.
- Cada camino de un nodo x hacia alguna hoja descendiente contiene el mismo número de nodos negros.

La última condición es esencial para garantizar que el árbol esté balanceado. Cuando alguno de los subárboles crece más que el otro (cuando se deja de cumplir esta propiedad) es necesario un rebalanceo a partir de "giros"(en sentido figurado o gráfico), de tal forma que el árbol resultante siga estando ordenado y sea rojinegro.

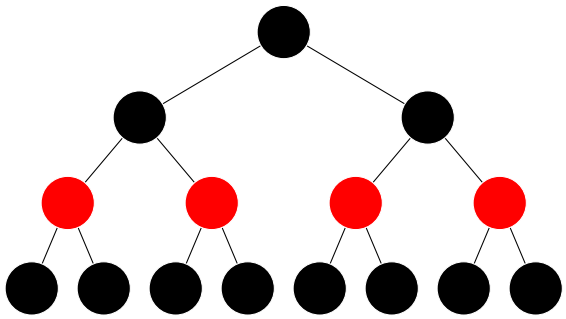


Figura 8.1: Representación gráfica de un árbol Rojinegro

Definición 8.1 (Definición inductiva de árboles Rojinegros)**.** *Los árboles Rojinegros de elementos de un conjunto A se definen como:*

- `Void` *es un árbol Rojinegro*
- *Si* $x \in A$ *y* t_1 t_2 *son árboles Rojinegros, entonces (*`Node` x t_1 t_2*) es un árbol Rojinegro.*
- *Son todos*

En donde `Void` es el constructor para el árbol vacío y `Node` construye un árbol con raíz y subárboles derecho e izquierdo.

La definición como tipo de dato algebraico es la siguiente:

Definición 8.2 (Tipo de dato `Rojinnegro`)**.**

```
data Color = Rojo | Negro

data Rojinegro a = Void
         | Node (a,Color) (Rojinegro a) (Rojinegro a)
```

Es necesario definir el tipo `Color` para representar el color de los nodos. La condición de coloración sí puede agregarse como parte del tipo de dato, sin embargo, la condición de orden no, por la misma razón que se vio en los árboles binarios de búsqueda. Definimos el álgebra semántica correspondiente

Definición 8.3 (Álgebra semántica de `Rojinegro`)**.** *Definición de álgebra semántica*

```
type AlgebraRN a r = (r, (a,Color) -> r -> r -> r)
```

Utilizando la definición 8.3 se implementa el operador `foldRN`

Definición 8.4 (Fold General)**.**

```
foldRN :: AlgebraRN a r -> Rojinegro a -> r
foldRN (b, f) = fold where
 fold Void          = b
 fold (Node x l r) = f x (fold l) (fold r)
```

Este tipo de árbol es un caso particular de los árboles con información en todos los nodos en donde cada nodo almacena el par que tiene la información del nodo y el color correspondiente.

8.2. Árboles de Huffman

Los árboles de Huffman son utilizados para cifrar un texto, con lo que se conoce como el Cifrado de Huffman que fue diseñado por el Científico de la Computación David A. Huffman mientras realizaba su doctorado en el MIT en 1952[27].

Este algoritmo recibe un texto claro[1] con el cual genera una tabla de cifrado en donde se asocia una cadena binaria a cada caracter que aparezca en el texto original.

Esta tabla es calculada a partir de la probabilidad estimada de que un caracter aparezca en el mensaje a cifrar, la estimación se hace con base en las apariciones de cada uno de los caracteres en la entrada, por lo que el cifrado de Huffman no es siempre el mismo y depende de la tabla generada en la primera parte del algoritmo[38].

Un árbol de Huffman es un árbol binario que almacena la frecuencia de cada letra en el texto utilizado para construirlo. Las hojas de un árbol de Huffman almacenan un caracter y el número de apariciones que tuvo este, mientras que los nodos internos guardan la suma de las apariciones de sus hojas descendientes teniendo en la raíz la longitud del texto original[39].

Para construir un árbol de Huffman se siguen los siguientes pasos[40]:

1. Se recorre el texto guardando la información de las frecuencias en una lista de pares (caracter y frecuencia).

[1] En criptografía el texto claro se refiere a un mensaje que no ha sido cifrado o codificado[37].

2. Por cada nodo de la lista, se crea una hoja del árbol de Huffman con la misma información del nodo correspondiente.
3. Se crea un *minHeap* utilizando la frecuencia como comparadores.
4. Se toman los dos caracteres con menor frecuencia del *heap*.
5. A partir de estos se crea un nuevo nodo que almacene la suma de las frecuencias y como subárboles las hojas seleccionados en el paso anterior.
6. Para los elementos restantes, se van agregando uno a uno como hojas del árbol hasta que el *heap* quede vacío.
7. El árbol de Huffman está completo.

Una vez obtenido el árbol se construye la tabla de cifrado, para esto se busca desde la raíz cada uno de los caracteres en el árbol y por cada paso en el camino para encontrarlo se agrega un binario (0 ó 1) al código que le corresponde, si el camino es por el hijo izquierdo se agrega un 0, en caso contrario se agrega un 1. De esta forma cada uno de los caracteres tiene un código único. Para cifrar un mensaje, se busca cada letra del mensaje en la tabla y se sustituye por el código binario que el corresponde[40].

La siguiente estructura es el árbol de Huffman generado con el texto *this is an example of a huffman tree* se puede ver en la figura 8.2.

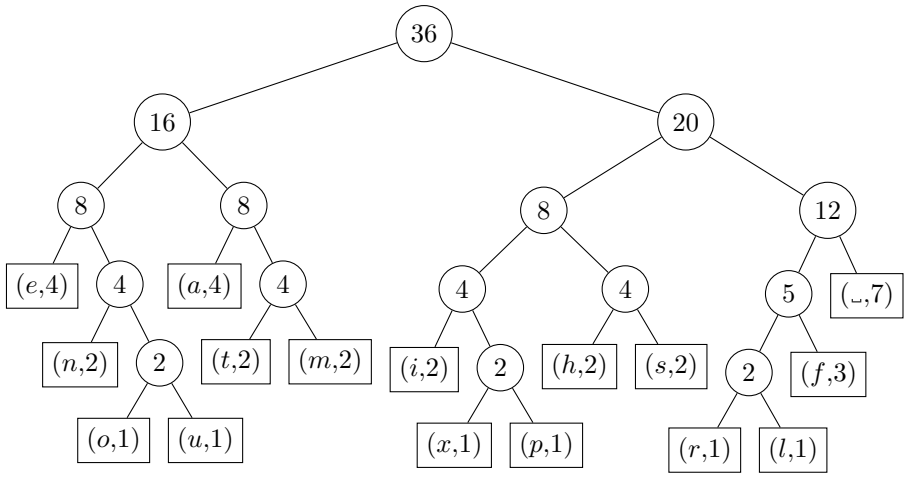


Figura 8.2: Representación gráfica del árbol de Huffman correspondiente al texto *this is an example of a huffman tree*

Caracter	Código
e	000
n	0010
o	00110
u	00111
a	010
t	0110
m	0111
i	1000
x	10010
p	10011
h	1010
s	1011
r	11000
l	11001
f	1101
␣	111

Cuadro 8.1: Tabla de cifrado del árbol de Huffman de la figura 8.2

Si se quisiera cifrar la palabra *Huffman* se haría de la siguiente manera:

H	u	f	f	m	a	n
1010	00111	1101	1101	0111	010	0010

El mensaje cifrado es **10100011111011101011101000010**

Definición 8.5 (Definición inductiva de árboles de Huffman)**.** *Un árbol de Huffman se define como sigue:*

- *Si a es un caracter y i un Entero, entonces (*`Leaf` *x i) es un árbol de Huffman*
- *Si i es un Entero y t_1, t_2 son árboles de Huffman, entonces (*`Node` *i t_1 t_2) es un arbol de Huffman.*
- *Son todos*

El constructor `Leaf` representa las hojas que almacenan un caracter con su frecuencia mientras que `Node` son los nodos internos del árbol que almacenan la suma de las frecuencias. La implementación en Haskell es como sigue:

Definición 8.6 (Tipo de dato HuffTree)**.**

```
type Par = (Char, Int)

data HuffTree = Leaf Par | Node Int HuffTree HuffTree
```

Primero se define un sinónimo para representar un par de un caracter con su frecuencia. Ahora se define el álgebra semántica correspondiente.

Definición 8.7 (Álgebra semántica de `HuffTree`)**.** *Definición de álgebra*

```
type HuffAlgebra t = (Par -> t, Int -> t -> t -> t)
```

Por último la implementación del operador `fold` con la función `foldHuff`.

Definición 8.8 (Fold General)**.**

```
foldHuff :: HuffAlgebra t -> HuffTree -> t
 foldHuff (f, g) = fold where
   fold (Leaf p)     = f p
   fold (Node i l r) = g i (fold l) (fold r)
```

Esta estructura es un caso particular de los árboles binarios con información en todas las hojas en donde cada nodo almacena una tupla que corresponde al carácter y su frecuencia. Y con el uso de los operadores de plegado se pueden escribir algunas de las funciones de cifrado.

8.3. Árboles B

Los árboles B son árboles de búsqueda autobalanceables. La idea central de los árboles B es mantener la altura del árbol lo más pequeña posible conservando el orden logarítmico de las operaciones básicas. Esto se logra teniendo nodos internos con un número variable de hijos dentro de un rango definido. Cuando se hace una inserción o se elimina un elemento el número de hijos puede cambiar por lo que los nodos pueden partirse o combinarse para conservar el número de hijos dentro del rango [41][42].

Dentro de cada nodo del árbol se almacenan llaves ordenadas dentro de un rango, los hijos deben preservar el orden. Los árboles B se mantienen balanceados siempre pues todas las hojas de este se encuentran al mismo nivel[43].

Estos árboles representan una ventaja en tiempo de consulta respecto a otras estructuras cuando la información almacenada no se tiene en la memoria principal y se tiene que hacer una consulta a disco, por cada nodo se tiene que hacer una consulta, lo cual en un árbol tradicional implicaría h consultas siendo

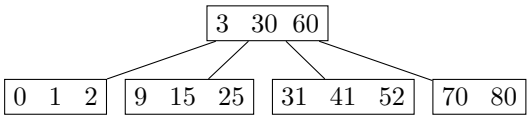


Figura 8.3: Representación gráfica de un árbol B

h la altura del árbol. Como el principal objetivo de los árboles B es mantener la altura mínima, el número de consultas disminuye[43].

Las propiedades de los árboles B son[41][43]:

- Todas las hojas se encuentran en el mismo nivel (a la misma altura).
- Un Árbol B se define en términos de un grado T
- Todos los nodos, excepto la raíz deben contener $T - 1$ llaves.
- La raíz contiene mínimo una llave.
- Todos los nodos deben tener máximo $2t - 1$ llaves.
- El número de hijos de cada nodo es igual $k + 1$ donde k es el número de llaves que contiene.
- Todas las llaves de un nodo están ordenadas en orden ascendente.
- El hijo del nodo entre las llaves k_1 y k_2 contiene todas las llaves entre k_1 y k_2

Los árboles B crecen en amplitud a diferencia de los árboles binarios de búsqueda autobalanceables que crecen en profundidad.

Definición 8.9 (Definición inductiva de árboles B)**.** *Un árbol B se define como sigue:*

- `BEmpty` *es un árbol B.*
- *Si* $i_1, i_1, ..., i_n$ *es un rango ordenado y* $b_1, b_2, ..., b_{n+1}$ *son árboles B, entonces (*`BNode` *[*$i_1, i_1, ..., i_n$*] [*$b_1, b_2, ..., b_{n+1}$*]) es un árbol B.*
- *Son todos*

Aquí se observa que `BEmpty` representa al árbol vacío y `BNode` construye arboles con n llaves y $n+1$ hijos. La implementación en Haskell de árboles B es la siguiente:

Definición 8.10 (Tipo de dato `B`)**.**

```
data B = BEmpty | BNode [Int] [B] Int
```

Es importante notar que el constructor `BNode` recibe un parámetro entero adicional, el cual representa la dimensión del árbol, esto es para simplificar la implementación en un lenguaje puramente funcional como lo es Haskell en donde no se tiene estado.

Se define ahora el álgebra semántica correspondiente al tipo de dato algebraico 8.10.

Definición 8.11 (Álgebra Semántica de `B`)**.** *Definición del álgebra semántica*

```
type AlgebraB b = (b, [Int] -> [b] -> Int -> b)
```

Por último se usa el álgebra recién definida para dar el operador `foldB`

Definición 8.12 (Fold General)**.**

```
foldB :: AlgebraB b -> B -> b
foldB (b,f) = fold where
 fold BEmpty = b
 fold (BNode ls bs i) = f ls (map fold bs) i
```

Esta estructura es un caso particular de las Rosadelfas vistas anteriormente en donde los elementos almacenados en los nodos son listas y el número de descendientes de los nodos no está acotado.

Capítulo 9

¿Cuándo una función es un fold?

Hasta este punto se ha usando la propiedad universal de cada operador `fold` para saber cuándo una función h puede reescribirse o no como una instancia de `fold`

$$h = \texttt{fold}\ f$$

Sin embargo, no es una forma efectiva de responder la pregunta ¿cuándo puede una función escribirse como un `fold`? pues es necesario conocer de antemano la función f que recibirá el operador de plegado como argumento y esto no es siempre una tarea sencilla. Al mismo tiempo, la propiedad universal nos proporciona en algunos casos una guía para construir una función f apropiada, pero esto tampoco es del todo convincente pues se trata de una heurística y no es tan sencillo de aplicar en la práctica[44].

Para encontrar una técnica eficaz y correcta para identificar las funciones que se pueden implementar con programación origami, se hace uso de la Teoría de álgebras, que se ha utilizado a lo largo de este trabajo, en conjunto con la Teoría de Categorías que está estrechamente relacionada con los operadores de plegado.

9.1. Teoría de Categorías

La Teoría de Categorías es una forma abstracta de tratar objetos matemáticos. La influencia de esta rama de las matemáticas ha sido bastante evidente en diferentes campos de las Ciencias de la Computación, tales como diseño de lenguajes de programación, modelos semánticos para lenguajes de programación, modelos de concurrencia, teoría de tipos, polimorfismo, teoría de autómatas,

diseño de algoritmos, entre muchas otras[45][46].

En la lista anterior se puede apreciar la generalidad de la Teoría de Categorías, pues se trata de un sistema conceptual y de notación básico en el mismo sentido que la teoría de conjuntos. La principal ventaja de esta generalidad es el alto nivel de abstracción de conceptos, lo que le permite ser versátil al modelar diferentes campos de la Computación[46][47].

Es por estas razones que la Teoría de Categorías es un modelo que nos servirá para abstraer la funcionalidad de los operadores de plegado, y así poder encontrar condiciones necesarias y suficientes para que una función pueda escribirse haciendo uso de ellos.

A continuación se definirán formalmente algunos conceptos básicos necesarios para el desarrollo de este capítulo (para más detalle, se pueden consultar algunas referencias de Teoría de Categorías con un enfoque a Ciencias de la Computación[46, 47]).

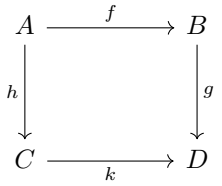
9.1.1. Conceptos Básicos

Intuitivamente la noción de categoría es muy simple, se trata de un conjunto de *objetos* y *flechas* que los relacionan, éstas reciben el nombre de morfismos[47]. Los morfismos son funciones que van de un objeto a otro, siendo la composición un concepto fundamental en la Teoría de Categorías. La definición formal de categoría es la siguiente[45][46][48]:

Definición 9.1 (Categoría)**.** *Una categoría C es un par $\langle \mathcal{O}, \mathcal{M} \rangle$, en donde $\mathcal{O}$ es una colección de objetos y $\mathcal{M}$ una colección de morfismos $f : A \to B$ para cada par de objetos A, B y una operación $\circ$ de composición con las siguientes propiedades:*

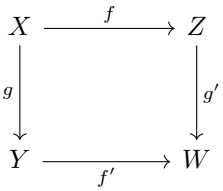
- **Dominio** *Si se tienen los morfismos $f : A \to B$ y $g : B \to C$ entonces $g \circ f$ es un morfismo del objeto A al objeto C.*
- **Asociatividad** *Para cualesquiera morfismos $f : A \to B$, $g : B \to C$ y $h : C \to D$ con A, B, C y D objetos, se cumple que $h \circ (g \circ f) = (h \circ g) \circ f$*
- **Identidad** *Para cada objeto $X \in C$, existe un morfismo identidad denotado Id_X, tal que $\mathsf{Id}_X : X \to X$.*
- **Neutro** *Para cada morfismo $f : A \to B$ se tiene que $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$*

La categoría de los objetos $\{A, B, C, D\}$ y los morfismos $f : A \to B$, $g : B \to D$, $h : A \to C$ y $k : C \to D$ se puede representar gráficamente como se muestra a continuación.



Definición 9.2 (Diagrama Conmutativo)**.** *En Teoría de Categorías, un diagrama conmutativo es un diagrama en el que todos los morfismos pararelos, que se obtienen a partir de la operación de composición concuerdan.*

Por ejemplo, consideremos el siguiente diagrama.



Decir que conmuta significa que $g' \circ f = f' \circ g$.

Otro concepto importante en Teoría de Categorías es el de funtor. Un funtor es un mapeo entre categorías, es decir, dadas dos categorías C y D, un funtor es una función entre objetos y entre flechas[47][46][45].

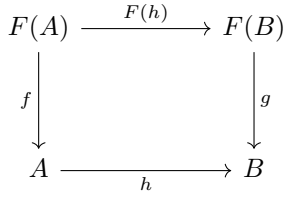
Definición 9.3 (Funtor)**.** *Dadas dos categorías C y D, un funtor F es una función que toma cada objeto X en C y lo mapea a un objeto $F(X)$ en D, y preserva los morfismos descritos en C, es decir, para todo morfismo $f \in C$ si $f : X \to Y \in C$ entonces $F(f) : F(X) \to F(Y) \in D$. Tal que cumple las siguientes propiedades:*

- **Identidad** *Para todo objeto $X \in C$ sucede que $F(\mathsf{Id}_X) = \mathsf{Id}_{F(X)}$*
- **Composición** *Para los morfismo en C, si $f : X \to Y$ y $g : Y \to Z$, entonces $F(g \circ f) = F(g) \circ F(f)$*

Los anteriores son los conceptos elementales de Teoría de Categorías. A continuación se definen nociones que están más relacionadas con el desarrollo de este trabajo, tales como álgebras y homomorfismos, éstos ya se han tratado en capítulos anteriores, sin embargo, ahora se darán definiciones centradas en la Teoría de Categorías.

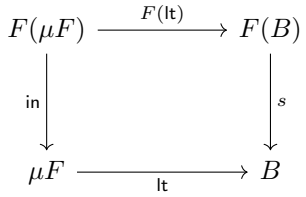
Definición 9.4 (F-álgebra)**.** *Dado un funtor $F : C \to C$ en una categoría C, una F-álgebra es un par $\langle A, f \rangle$ tal que $f : F(A) \to A$.*

Definición 9.5 (Homomorfismo)**.** *Un homomorfismo $h : \langle A, f\rangle \to \langle B, g\rangle$, de un álgebra en otra, es un morfismo $h : A \to B$ tal que el siguiente diagrama conmuta:*



es decir, se cumple que $h \circ f = g \circ F(h)$

Definición 9.6 (Álgebra Inicial)**.** *Una álgebra inicial es un objeto inicial en la categoría con F-álgebras como objetos y homomorfismos como flechas, es decir, que existe un único morfismo* lt *del álgebra inicial a cualquier otra álgebra $\langle B, s\rangle$. Si existe, el álgebra inicial es única y se denota como $\langle \mu F, \mathsf{in}\rangle$.*

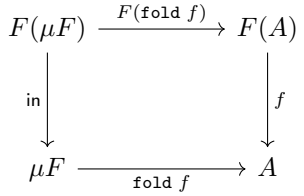


Con estos conceptos se puede definir el tratamiento categórico de `fold`.

9.2. Tratamiento categórico de fold

En esta sección se define el tratamiento categórico del operador de plegado `fold` a partir de los conceptos tratados en la sección anterior. Esto nos sirve para abstraer el comportamiento de `fold` con el uso de Teoría de Categorías y así poder describir condiciones necesarias y suficientes para saber si una función puede definirse como instancia de este operador. Esta abstracción está fuertemente determinada por la interpretación computacional de un álgebra inicial como un tipo de datos.

Como se vio en la definición 9.6 de la sección anterior, una álgebra inicial $\langle \mu F, \mathsf{in}\rangle$ tiene un único homomorfismo $h : \langle \mu F, \mathsf{in}\rangle \to \langle A, f\rangle$ del álgebra inicial a cualquier otra álgebra $\langle A, f\rangle$. Ese único homomorfismo h es justamente la función `fold` f, es decir, `fold` f se define como la única flecha que hace que el siguiente diagrama conmute[44][49]:



Para poder comprender mejor la función `fold` en términos de álgebras iniciales, recordemos la sección 1.5 en donde se definió el concepto de álgebra semántica que se ha utilizado a lo largo de este trabajo, vimos que se podía definir una álgebra semántica especial usando exactamente los tipos de los constructores, ésta corresponde al álgebra inicial.

Como ejemplo, consideremos que tenemos una lista de elementos de tipo `Bool`, entonces se puede definir el álgebra inicial de esta estructura en Haskell como sigue:

```
type iniAlg = ([Bool], Bool -> [Bool] -> [Bool])
```

Es decir, para este tipo de lista en particular, el tipo de la lista vacía es `[Bool]` y el tipo de la función `cons` es `Bool->[Bool]->[Bool]`. El álgebra semántica definida para listas (definición 2.4) parametriza los tipos usando variables de tipo para dar una definición más general, por ejemplo se puede definir la siguiente álgebra para calcular la longitud de la lista:

```
lenAlg :: (Int, Bool -> Int -> Int)
lenAlg =  (0, (\x y -> 1 + y))
```

Y de hecho `foldList lenAlg` calcula correctamente la longitud de la lista. En otras palabras, el operador `foldList` transforma el álgebra inicial (`iniAlg`) en una nueva álgebra (`lenAlg`)[47][50].

Así como el operador `fold` tiene una correspondencia en la Teoría de Categorías, sus propiedades pueden definirse también en términos de álgebras iniciales. La propiedad universal se define categóricamente como sigue.

Definición 9.7. *(Propiedad Universal) La propiedad universal del operador de plegado* `fold` *con tratamiento categórico se expresa con la siguiente equivalencia:*

$$h = \texttt{fold}\ f \iff h \circ \texttt{in} = f \circ Fh$$

La ida ($\Rightarrow$) de la equivalencia expresa que `fold` f es un homomorfismo desde el álgebra inicial $\langle \mu F, \mathsf{in} \rangle$ hacía otra álgebra $\langle A, f \rangle$. Del otro lado ($\Leftarrow$) de la equivalencia determina que cualquier otro homomorfismo h entre estas dos álgebras debe ser igual a `fold` f. Esto quiere decir que la propiedad universal expresa que `fold` f es el único homomorfismo desde $\langle \mu F, \mathsf{in} \rangle$ hacia $\langle A, f \rangle$[44][51].

9.3. Identificar funciones que pueden escribirse en origami

Con este nuevo tratamiento se puede observar que el operador `fold` define un morfismo del tipo $\mu F \to A$. Entonces la pregunta que da título a este capítulo puede replantearse como ¿Cuándo un morfismo arbitrario $h : \mu F \to A$ puede escribirse de la forma $h = \texttt{fold}\ f$ para algún otro morfismo $f : FA \to A$?[44].

En esta sección, se responderá a esta pregunta para el caso especial de la categoría SET, en donde los morfismos son funciones totales entre conjuntos, es decir, funciones definidas para todos los elementos del dominio.Primero se dan algunas definiciones, teoremas y lemas que serán necesarios.

Definición 9.8 (Kernel)**.** *El kernel de una función $f : A \to B$ es el conjunto de pares de elementos de A tales que al evaluar f con ellos arrojan el mismo resultado. Es decir*

$$\texttt{ker}\ f = \{(x, y) \in A \times A \mid f\ x = f\ y\}$$

Lema 9.1. *Sean $f : A \to B$ y $h : A \to C$ dos morfismos. Existe al menos un morfismo de B en C y el kernel de f está contenido en el kernel de h sí y solo si se puede definir un morfismo $g : B \to C$ tal que h es la composición de g con f, es decir, se cumple la siguiente equivalencia:*

$$(\exists g : B \to C \text{ tal que } h = g \circ f) \iff (\texttt{ker}\ f \subseteq \texttt{ker}\ h \wedge B \to C \neq \varnothing)$$

Demostración. Primero probemos la ida ($\Rightarrow$) sabemos que existe $g : B \to C$ por lo tanto $B \to C \neq \varnothing$, ahora:

$$\begin{array}{cc} & (x, y) \in \texttt{ker}\ f \\ \iff & \{\text{Definición de Kernel}\} \\ & f\ x = f\ y \\ \iff & \{\text{Aplicando } g\} \\ & g(f\ x) = g(f\ y) \\ \iff & \{h = g \circ f\} \\ & h\ x = h\ y \\ \iff & \{\text{Definición de Kernel}\} \\ & (x, y) \in \texttt{ker}\ h \end{array}$$

Para el regreso ($\Leftarrow$) hay dos casos

Caso 1: $B = \varnothing$

Cuando $B = \varnothing$ entonces g es la única función existente en $B \to C$, la vacía. Por lo tanto $g \circ f$ es también vacía.

Por otro lado A tiene que ser $\varnothing$ por el tipo de f, entonces h es también vacía y por lo tanto $h = g \circ f$.

Caso 2: $C \neq \varnothing$

Se define la función g de la siguiente forma:

$$g\ b = \begin{cases} h\ a, \text{ para alguna } a \text{ tal que } f\ a = b, & \text{si } b \text{ está en el rango de } f \\ c, \text{ con } c \text{ una constante arbitraria} & \text{en otro caso} \end{cases}$$

Esta definición es correcta pues si hubiera más de una opción a y a' tales que $f\ a = b = f\ a'$ entonces $h\ a = h\ a'$ pues se tiene como hipótesis que $\texttt{ker}\ f \subseteq \texttt{ker}\ h$. Por la forma en que g fue construida, se tiene que $h\ a = g\ (f\ a)$ para toda a [44].

$\therefore$ se puede concluir que $h = g \circ f$. □

Lema 9.2. *Si existe un morfismo $f : \mu F \to A$, entonces existe también el morfismo $g : FA \to A$, es decir, la siguiente implicación es cierta:*

$$\mu F \to A \neq \varnothing \Rightarrow FA \to A \neq \varnothing$$

Demostración. Notemos que $FA \to A \neq \varnothing$ es equivalente a $A = \varnothing \Rightarrow FA = \varnothing$ [44].

$$\begin{array}{cc} & A = \varnothing \\ \Rightarrow & \{\mu F \to A \neq \varnothing\} \\ & \mu F = \varnothing \\ \Rightarrow & \{\mathsf{in} : F(\mu F) \to \mu F\} \\ & F(\mu F) = \varnothing \\ \Rightarrow & \{\mu F = \varnothing = A\} \\ & FA = \varnothing \end{array}$$

□

Teorema 9.9. *Supongamos $h : \mu F \to A$. Entonces $h = \texttt{fold}\ f$, si y sólo sí,* $\texttt{ker}\ h$ *es congruente bajo* in*, esto es, que el kernel de Fh está contenido en el kernel de h compuesto con* in*, es decir, la siguiente equivalencia es cierta:*

$$(\exists f : FA \to A \text{ tal que } h = \texttt{fold}\ f) \iff (\texttt{ker}(Fh) \subseteq \texttt{ker}(h \circ \mathsf{in}))$$

Demostración.

$$\begin{array}{cc} & \exists f : FA \to A \text{ tal que } h = \texttt{fold}\ f \\ \iff & \{\text{Propiedad Universal 9.7}\} \\ & \exists f : FA \to A \text{ tal que } h \circ \mathsf{in} = g \circ Fh \\ \iff & \{\text{Lema 9.1}\} \\ & \texttt{ker}(\texttt{Fh}) \subseteq \texttt{ker}(\texttt{h} \circ \mathsf{in} \wedge \texttt{FA} \to \texttt{A} \neq \varnothing) \\ \iff & \{\text{Lema 9.2}\} \\ & \texttt{ker}(Fh) \subseteq \texttt{ker}(h \circ \mathsf{in}) \end{array}$$

□

El teorema anterior define la condición necesaria y suficiente para determinar si un morfismo $h : \mu F \to A$ puede escribirse como `fold` f para algún morfismo $f : FA \to A$.

Es importante observar que toda la teoría desarrollada en este capítulo no corresponde a una estructura de datos específica, si no que se trata de un razonamiento generalizado sobre los operadores de plegado, por lo que es posible utilizar el resultado obtenido en todas las estructuras vistas hasta este punto.

9.4. Poniendo en práctica la condición

En esta sección se retoman algunos ejemplos vistos en capítulos anteriores para mostrar en la práctica el teorema 9.9.

9.4.1. Listas

Para las listas finitas de elementos de algún tipo `a`, el teorema 9.9 nos dice que una función h se puede reescribir usando el operador `foldList` cuando las listas que h recibe como argumento son cerradas bajo el constructor `Cons` (`:`), es decir, cuando se cumple la siguiente implicación[44]:

$$h\ \texttt{xs} = h\ \texttt{ys} \Rightarrow h\ (\texttt{x} : \texttt{xs}) = h\ (\texttt{x} : \texttt{ys})$$

Tomemos como ejemplo la función `map`:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

Probemos ahora la cerradura de `map` sobre el constructor `Cons`.

$$\begin{array}{rc} & \texttt{map}\ f\ (x : xs) = \texttt{map}\ f\ (x : ys) \\ \iff & \{\ \text{Definición de } \texttt{map}\ \} \\ & (f\ x) : \texttt{map}\ f\ xs = (f\ x) : \texttt{map}\ f\ ys \\ \Rightarrow & \{\ \text{Hipótesis}\ \} \\ & (f\ x) : \texttt{map}\ f\ xs = (f\ x) : \texttt{map}\ f\ xs \end{array}$$

$\therefore$ `map` puede escribirse como una instancia de `foldList`.

En contraste la función `stail` que regresa la cola de una lista de forma segura, es decir, sin lanzar errores, no puede escribirse como un `foldList`.

```
stail :: [a] -> [a]
stail []     = []
stail (x:xs) = xs
```

Basta con probar que `stail` no es cerrada bajo el constructor `Cons`, para eso se muestra un contraejemplo.

Consideremos las siguientes listas `xs = []` y `ys = [0]`, se puede observar que `stail xs = stail ys`, ambas regresan la lista vacía. Sin embargo, `stail (0:xs)` $\neq$ `stail (0:ys)`, ya que `stail (0:xs) = []` mientras que `stail (0:ys) = [0]`.

$\therefore$ `stail` no puede escribirse como `foldList`.

9.4.2. Árboles binarios con información sólo en las hojas

En el caso de los árboles binarios que solo almacenan información en las hojas, el teorema 9.9 se reduce a decir que una función arbitraria h que toma un árbol de tipo `Tree a` y regresa un valor de algún tipo `b`, puede escribirse directamente como un `foldTree` cuando el árbol que recibe h es cerrado bajo el constructor `Node`[44]. Es decir, que para cualesquiera árboles l y r se cumple que:

$$h\ l = h\ l' \wedge h\ r = h\ r' \Rightarrow h\ (\texttt{Node}\ l\ r) = h\ (\texttt{Node}\ l'\ r')$$

Veamos como ejemplo la función `tam` que calcula el tamaño de un árbol.

```
tam :: Tree a -> Int
tam (Leaf e)   = 1
tam (Node l r) = (tam l) + (tam r)
```

Probemos la cerradura. Para cualesquiera árboles `l`, `r`, `l'` y `r'`, tales que `tam l` $=$ `tam l'` y `tam r` $=$ `tam r'`.

$$\begin{array}{rc} & \texttt{tam}\ (\texttt{Node}\ \texttt{l}\ \texttt{r}) = \texttt{tam}\ (\texttt{Node}\ \texttt{l}'\ \texttt{r}') \\ \iff & \{\ \text{Definición de } \texttt{tam}\ \} \\ & (\texttt{tam}\ \texttt{l}) + (\texttt{tam}\ \texttt{r}) = (\texttt{tam}\ \texttt{l}') + (\texttt{tam}\ \texttt{r}') \\ \Rightarrow & \{\ \text{Hipótesis}\ \} \\ & (\texttt{tam}\ \texttt{l}) + (\texttt{tam}\ \texttt{r}) = (\texttt{tam}\ \texttt{l}) + (\texttt{tam}\ \texttt{r}) \end{array}$$

$\therefore$ la función `tam` puede escribirse como instancia de `foldTree`.

Consideremos ahora la función `bal` que recibe un árbol y regresa un valor de tipo `Bool` que dice si todas las hojas del árbol se encuentran en el mismo nivel. Tomemos los siguientes árboles para exhibir un contraejemplo:

```
t1 = (Node (Leaf 0) (Leaf 0))
t2 = (Leaf 0)
```

En ambos casos la función `bal` devuelve verdadero, pues las hojas están en el mismo nivel, entonces `bal t1` $=$ `bal t2`.

Pero `bal` (`Node t1 t1`) $\neq$ `bal` (`Node t1 t2`) ya que la primera llamada a `bal` regresa `True` mientras que la segunda regresa `False`.

$\therefore$ `bal` no puede escribirse como una instancia de `foldTree`.

9.4.3. Árboles binarios con información en todos los nodos

Con los árboles binarios que almacenan información también en los nodos internos, el teorema 9.9, dice que una función h que toma un árbol de tipo `BinT a` y regresa un valor de algún tipo `b` se puede reescribir como instancia del operador de plegado definido para esta estructura `foldBin` si h es cerrada bajo el constructor `Node`. Es decir, que la siguiente implicación es cierta para cualesquiera árboles l y r:

$$h\ l = h\ l' \wedge h\ r = h\ r' \Rightarrow h\ (\texttt{Node}\ e\ l\ r) = h\ (\texttt{Node}\ e\ l'\ r')$$

En donde e es un nuevo elemento del árbol.

Como ejemplo, probemos que la función que aplana el árbol en una lista haciendo un recorrido *in order* puede definirse como un `fold`.

```
inorder :: BinT a -> [a]
inorder Void         = []
inorder (Node x l r) = (inorder l) ++
                       x:(inorder r)
```

Sean `l`, `l'`, `r` y `r'` árboles de tipo `Bin a` tales que `inorder l` = `inorder l'` y `inorder r` = `inorder r'` y `x` un elemento de tipo `a`.

$$\begin{aligned}
& \texttt{inorder}\ (\texttt{Node x l r}) = \texttt{inorder}\ (\texttt{Node x l' r'}) \\
\iff & \quad \{\ \text{Definición de } \texttt{inorder}\ \} \\
& (\texttt{inorder l}) + +\texttt{x} : (\texttt{inorder r}) = (\texttt{inorder l'}) + +\texttt{x} : (\texttt{inorder r'}) \\
\Rightarrow & \quad \{\ \text{Hipótesis}\ \} \\
& (\texttt{inorder l}) + +\texttt{x} : (\texttt{inorder r}) = (\texttt{inorder l}) + +\texttt{x} : (\texttt{inorder r})
\end{aligned}$$

$\therefore$ `inorder` es una función que puede escribirse con el operador `foldTree`.

Ahora veamos el caso de la función `remove` que elimina un elemento de un árbol. Esta función no puede escribirse como un `fold`, veamos un contraejemplo que lo prueba. Consideremos los siguientes árboles:

```
t1 = Void
t2 = (Node 0 Void Void)
```

con estos árboles las llamadas `remove 0 t1` y `remove 0 t2` regresan el árbol vacío, es decir, `remove 0 t1 = remove 0 t2`.
Sin embargo `remove 0` (`Node 0 t1 t1`) $\neq$ `remove 0` (`Node 0 t1 t2`) pues mientras

que la primera llamada regresa `Void`, la segunda regresa el árbol de un sólo nodo con el valor `0`.

$\therefore$ `remove` no puede escribirse como instancia de `foldBin`

En contraste la función `removeAll` que elimina todas las apariciones de un elemento en un árbol, si es cerrada bajo el constructor `Node`, por lo que puede reescribirse utilizando la técnica de programación origami. La demostración es muy similar a la presentada anteriormente sobre la función `inorder`.

9.4.4. Rosadelfas

Por último veamos el caso de las Rosadelfas, para las cuales el teorema 9.9 indica que una función h que toma un elemento de tipo `Rosadelfa` puede implementarse como una instancia de `foldRosa` si la rosadelfa recibida por h es cerrada bajo el constructor `Node`, es decir que se cumple la siguiente implicación:

$$\bigwedge_{i=0}^{n}(h\ t_i = h\ t'_i) \Rightarrow h\ (\texttt{Node}\ x\ [t_0 \ldots t_n]) = h\ (\texttt{Node}\ x\ [t'_0 \ldots t'_n])$$

Probemos que la función `nodos`, que calcula el número de nodos de una rosadelfa, puede escribirse como una instancia de `foldRosa`.

```
nodos :: Rosadelfa a -> Int
nodos Void       = 0
nodos (Node e l) = 1 + sum (map nodos l)
```

Sean $\texttt{t}_0 \ldots \texttt{t}_\texttt{n}$ una colección de rosadelfas que almacenan elementos de tipo `a`, tales que $\bigwedge_{i=0}^{n}(h\ \texttt{t}_\texttt{i} = h\ \texttt{t}'_\texttt{i})$. Y sea `x` un elemento de tipo `a`.

$$\begin{array}{ll}
 & \texttt{nodos (Node x [t}_0 \ldots \texttt{t}_\texttt{n}\texttt{])} = \texttt{nodos (Node x [t}'_0 \ldots \texttt{t}'_\texttt{n}\texttt{])} \\
\iff & \{\text{ Definición de } \texttt{nodos} \} \\
 & 1 + \texttt{sum (map nodos [t}_0 \ldots \texttt{t}_\texttt{n}\texttt{])} = 1 + \texttt{sum (map nodos [t}'_0 \ldots \texttt{t}'_\texttt{n}\texttt{])} \\
\iff & \{\text{ Aplicación de } \texttt{map} \} \\
 & 1 + \texttt{sum [nodos t}_0 \ldots \texttt{nodos t}_\texttt{n}\texttt{]} = 1 + \texttt{sum [nodos t}'_0 \ldots \texttt{nodos t}'_\texttt{n}\texttt{]} \\
\Rightarrow & \{\text{ Hipótesis } \} \\
 & 1 + \texttt{sum [nodos t}_0 \ldots \texttt{nodos t}_\texttt{n}\texttt{]} = 1 + \texttt{sum [nodos t}_0 \ldots \texttt{nodos t}_\texttt{n}\texttt{]}
\end{array}$$

$\therefore$ la función `nodos` puede escribirse directamente usando el operador `foldRosa`.

Ahora probemos que la función `elimina`, que elimina un elemento de una rosadelfa, no es cerrada bajo el constructor `Node`. Tomemos las siguientes rosadelfas:

```
r1 = Void
r2 = Node 5 []
```

Se puede observar que `elimina 5 r1` = `elimina 5 r2`, pues ambas devuelven `Void`. Sin embargo, `elimina` (`Node 5 [r1]`) ≠ `elimina` (`Node 5 [r2]`), ya que la primera llamada regresa `Void` mientras que la segunda devuelve la rosadelfa con un sólo elemento.

∴ `elimina` no puede escribirse como instancia de `forlRosa`.

En conclusión, en este capítulo se mostró una condición necesaria y suficiente para reconocer las funciones que se pueden escribir como `fold`, este resultado es suficientemente general para poder utilizarse con las estructuras de datos vistas en los capítulos anteriores, gracias al uso de Teoría de Categorías y al tratamiento de los operadores de plegado dentro de este formalismo matemático. A parte de tratarse de un resultado interesante desde el punto de vista teórico, también tiene un uso práctico muy importante.

Es relevante señalar que el uso de esta técnica para reconocer las funciones que son instancias de `fold`, es menos complejo y más eficiente respecto al uso de la Propiedad Universal. Sin embargo, ésta es una técnica de decidibilidad, es decir, solo nos dice si la función puede escribirse o no usando el operador `fold` pero no nos dice cómo hacerlo.

Esta técnica ha sido extendida también hacia el operador `unfold`, si el lector está interesado puede consultar la referencia [44].

Conclusiones y trabajo futuro

Conclusiones

Este trabajo comenzó recopilando los estudios que se han hecho sobre los operadores de plegado y la programación origami con listas, esta investigación ya ha sido ampliamente desarrollada por diferentes autores y con diferentes enfoques como se vio en el capítulo 2. El objetivo principal fue extender esta teoría para estructuras más complejas, eligiendo árboles por el gran uso que tienen dentro de Ciencias de la Computación, generalizando los aspectos relevantes en el estudio de programación origami en listas hacia diferentes estructuras arbóreas. Las propiedades principales de los operadores de plegado estudiadas fueron la Propiedad Universal que define la expresividad de los operadores `fold` y el Principio de Fusión que permite componer funciones utilizando un único `fold` para hacer mas compacta una definición.

Se tomó como punto de partida para desarrollar este trabajo, en la capítulo 3, los árboles binarios con información sólo en las hojas, pues se trata de una estructura arbórea sencilla que sirvió para ejemplificar las definiciones y usos de los operadores de plegado en un contexto ajeno a las listas.

Después se presentó una clasificación de estructuras más compleja, los árboles binarios con información en todos los nodos. Para esto se mostraron dos ejemplos prácticos de usos de este tipo de árboles:

- Árboles binarios de búsqueda en el capítulo 5
- Árboles Cartesianos en el capítulo 6

Para ambos casos se dio una implementación de estas estructuras y las funciones definidas sobre ellas utilizando la técnica de programación origami. Aquí es importante mencionar que no se pueden dar implementaciones puramente origami, es decir, definidas en su totalidad como instancias de `fold`, esto se debe

a que no todas las funciones siguen el patrón encapsulado por los operadores de plegado y no pueden reescribirse haciendo uso de éstos.

La última clasificación de árboles definida fueron los árboles n-arios, es decir árboles que no limitan el número de descendientes que puede tener cada nodo, esta parte es especialmente interesante pues se utilizan listas para definir la estructura Rosadelfa en el capítulo 7, y cada nodo tiene una lista de árboles (sus descendientes), la función `fold` correspondiente hace uso de la función `map` de listas para hacer la recursión sobre todos los hijos de un nodo, de esta forma se combinan las definiciones de listas y árboles mostrando la "elegancia" de la programación funcional con operadores de plegado.

Para todos los árboles definidos en estas partes del trabajo se definió también la función `unfold` correspondiente que es una especie de inverso para el operador `fold`.

Después se definieron estructuras arbóreas avanzadas en el capítulo 8, las estructuras estudiadas fueron:

- Árboles Rojinegros
- Árbole de Huffman
- Árboles B

Estos árboles son casos particulares de las clasificaciones que ya se habían estudiado por lo que solo se definió la estructura y su operador de plegado.

En la parte final del trabajo se estudio el tratamiento categórico de los operadores de plegado, es decir, las definiciones de éstos dentro la Teoría de Categorías. Esto permitió definir una condición necesaria y suficiente para poder reconocer las funciones que pueden escribirse en términos de la función `fold` de forma eficiente y correcta. Este resultado es general para cualquier tipo inductivo y se mostró también como se comporta con las estructuras previamente definidas.

El producto final de este trabajo fue justamente el que se planteó al comienzo del mismo, obteniendo operadores de plegado para cada una de las estructuras definidas y probando la Propiedad Universal y el Principio de Fusión en cada caso. Esto resulta de mucha utilidad cuando se desarrollan estructuras de datos puramente funcionales pues se tiene una forma eficiente de iterar sobre ellas a través de la función `fold` correspondiente.

Trabajo Futuro

Uno de los temas que resultó de más interés durante la elaboración de este trabajo es el operador `unfold`, pues como bien lo dice Jeremy Gibbons en *The*

under-appreciated unfold [25], un operador que no ha sido tan estudiado como lo ha sido `fold` y no se le ha dado el lugar que merece (de ahí el título del artículo). Como trabajo a futuro sería interesante desarrollar la teoría propia de `unfold` como la que se ha desarrollado para `fold`, estudiar algunas de las propiedades que tiene este operador, su poder de abstracción o escritura y también los usos que se le podrían dar.

Se puede complementar la parte teórica de este trabajo con la Teoría de Categorías para estudiar más a fondo el fundamento teórico de los operadores de plegado.

Así como el operador `fold` encapsula el patrón mas común en la definición de funciones de una estructura de datos, sería interesante estudiar otros patrones que también son muy utilizados como operadores de plegado, como sucede con el operador `foldl` para listas. El siguiente es un operador de plegado alternativo para árboles:

```
foldt :: (a -> [b] -> b) -> b -> BinT a -> b
foldt _ v Void = v
foldt f v (Node x l r) = f x $ map (foldt f v) [l,r]
```

que recorre un árbol binario de forma transversal, construyendo una lista por cada nivel del árbol.

Otro aspecto interesante en el que puede ahondar es llevar la generalización que se obtuvo en este trabajo a estructuras más complejas como puede ser el caso de las gráficas. En estas estructuras la presencia de operadores de plegado podría resultar en optimizaciones sobre las funciones propias de estas estructuras. Esto resultaría muy útil pues los operadores de plegado son sencillos de definir y utilizar en la mayoría de los casos en comparación con otros métodos de iteración, así como para abstraer y compactar las definiciones de funciones.

Bibliografía

[1] Jeremy Gibbons. Origami programming. *The Fun of Programming*, pages 41–60, 2001. (document), 2

[2] Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008. (document), 3, 4, 5

[3] Favio E. Miranda Perea y Lourdes Del Carmen González Huesca. Curso de programación funcional y lógica. Notas de clase, Facultad de Ciencias, Universidad Nacional Autónoma de México, 2011. 1.1, 1.3, 1.3.1

[4] B. Jack Copeland. The church-turing thesis. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2020 edition, 2020. 1.2

[5] Richard Bird. *Thinking Functionally with Haskell*. University of Oxford, 2015. 1.2

[6] John C. Mitchell. *Concepts in Programming languages*. Cambridge University Press, primera edition, 2002. 1.2

[7] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. 1.2

[8] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Cambridge University Press, segunda edition, 2017. 1.2

[9] *Documentación de Haskell (`https://www.haskell.org/documentation`)*, 2010 (consultado en Febrero 3, 2020). 1.2, 2.1, 5.3.1

[10] Simon Peyton Jones Philip Wadler Paul Hudak, John Hughes. A history of haskell: Being lazy with class. *ACM*, 12:1–55, 2006. 1.2

[11] Graham Hutton. *Programming in Haskell*. segunda edition, 2016. 1.2, 1.3

[12] Leonidas Fegaras. Functional languages and higher-order functions. Class slides, University of Texas at Arlington, 2005. 1.3.1

[13] Assia Mahboubi. Inductive data types. Class slides, École polytechnique, 2011. 1.4

[14] Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Functional Programming*, pages 355–372, 1999. 1.4, 2, 2.3, 2.2.1, 2.1, 2.1

[15] Narciso Marti Oliet, Yolanda Ortega Mallén, and José Alberto Verdejo López. *Estructuras De Datos Y Métodos Algorítmicos.* Pearson, 2004. 4, 1.6

[16] Andrea Asperti and Giuseppe Longo. *Categories Types and Structures.* The MIT Press, 1991. 1.5

[17] Manuel Alcino Cunha. Recursion patterns as hylomorphisms. *Program Understanding and Re-engineering: Calculi and Applications*, pages 3–24, 2003. 1.5

[18] Johan Jeuring and S. Doaitse Swierstra. Grammars and parsing. 2001. 1.5, 1.5

[19] Claude Berge. *Graphs and Hypergraphs.* Elsevier Science Ltd, 1985. 1.6

[20] José É. Gallardo Ruiz. *Funciones de Plegado sobre listas.* Lección Magistral para el concurso a la plaza de TEU, 2000. 2

[21] Richard Bird. *Introducction to Functional Programming using Haskell.* University of Oxford, 1998. 2, 7, 7.1

[22] Lourdes Del Carmen González Huesca, Favio Ezequiel Miranda Perea, and Araceli Liliana Reyes Cabello. Operadores de plegado en programación funcional. *XLIII Congreso Nacional SMM*, 2010. 2, 2.1

[23] Favio E. Miranda Perea y Elisa Viso Gurovich. *Matemáticas Discretas.* Prensas de Ciencias, segunda edition, 2016. 2.1, 4.1

[24] Moses Schönfinkel. Über die bausteine der mathematischen logik. *Math. Ann*, 92:305–316, 1924. 1

[25] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. *ACM SIGPLAN Notices*, 34:1–6, 10 1999. 2.4, 2.4, 9.4.4

[26] Conal Elliott. Folds and unfolds all around us. Class slides, Tabula, 2013. 2.4

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 2009. 5, 8.2

[28] Guy Blelloch. *Parallel and Sequential Data Structures and Algorithms.* Universidad de Carnegie Mellon, 2015. 5.1

[29] Canek Peláez. *Estructuras de datos con Java moderno*. Prensas de Ciencias, primera edition, 2018. 5.1, 8.1

[30] *Documentación de Liquid Haskell (https://ucsd-progsys.github.io/liquidhaskell-blog/)*, 2012 (consultado en Febrero 13, 2020). 5.4.1

[31] Niki Vazou Ranjit Jhala, Eric Seidel. *Programming with Refinement Types An Introduction to LiquidHaskel*. segunda edition, 2017. 5.4.1

[32] *Documentación de Agda (https://agda.readthedocs.io/en/v2.6.1/)*, 2020 (consultado en Febrero 13, 2020). 5.4.1

[33] *Documentación de Coq (https://coq.inria.fr/)*, 2019 (consultado en Febrero 13, 2020). 5.4.1

[34] Niki Vazou Ranjit Jhala, Eric Seidel. *Programming with Refinement Types*. 2017. 5.6

[35] Jean Vuillemin. A unifying look at data structures. *Communications of ACM*, 23(4):229–239, 1980. 6, 6.1

[36] Andrew J. Kennedy. Drawing trees. *J. Functional Programming*, 6:527–534, 1996. 7

[37] José Galaviz Casas. *Introducción a la teoría de códigos y de la información*. Prensas de Ciencias, primera edition, 2012. 1

[38] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 8.2

[39] Jan Van Leeuwen. On the construction of huffman trees. *ICALP*, pages 382–410, 1976. 8.2

[40] Philip Wadler Richard Bird. *Introduction to functional programming*. PRENTICE HALL, primera edition, 1988. 8.2, 8.2

[41] Marc van Kreveld. Computational geometry, lecture slides para el curso geometric algorithms (`https://comptag.github.io/teaching-compGT/assets/snapshots/uunl-cs-infoga-17.pdf`). *University of Florida*, 2016. 8.3

[42] Chris Okasaki. *Purely Functional Data Structures*. Carnegie Mellon University, 1996. 8.3

[43] Mark Overmars Mark de Berg, Marc van Kreveld. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008. 8.3

[44] Thorsten Alterkirch Jeremy Gibbons, Graham Hutton. When is a function a fold or an unfold? *Theorical Computer Science*, 44:1–14, 2001. 9, 9.2, 9.2, 9.3, 9.3, 9.3, 9.4.1, 9.4.2, 9.4.4

[45] Lourdes del Carmen González Huesca. *Coinducción: de la Teoría de Categorias a la Programación Funcional.* Tesis de Licenciatura, Facutad de Ciencias, UNAM, 2007. 9.1, 9.1.1, 9.1.1

[46] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, 1991. 9.1, 9.1.1, 9.1.1

[47] Bartosz Milewski. *Cathegory Theory for Programmers.* Creative Commons, primera edition, 2019. 9.1, 9.1.1, 9.1.1, 9.2

[48] Varmo Vene. *Categorical Programming with Inductive and Coinductive Types.* Dissertation accepted for public defense of the degree of Doctor of Philosophy.Faculty of Mathematics, University of Tartu, Estonia, 2006. 9.1.1

[49] Diego Pedraza López. *Teoría de Categorías y Programación Funcional.* Universidad de Sevilla, 2018. 9.2

[50] Jan-Willem Buurlage. *Categories and Haskell An introduction to the mathematics behind modern functional programming.* Notes on Category Theory and Haskell, 2018. 9.2

[51] G. Malcolm. Algebraic data types and program transformation. *Science of Computer Programming*, 14(2-3):255–280, 1990. 9.2