UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE
INGENIERÍA

Estudio del lenguaje LiquidHaskell

Informe de Proyecto de Grado presentado por

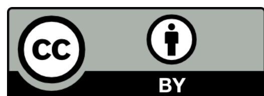
Felipe de León

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Marcos Viera
Alberto Pardo

Montevideo, 2 de diciembre de 2024



Estudio del lenguaje LiquidHaskell por Felipe de León tiene licencia CC Atribución 4.0.

Agradecimientos

Este trabajo no hubiera sido posible sin la contribución de muchas personas que me han acompañado en mi paso por la facultad de ingeniería. Quiero agradecerles a todos, ya que la culminación de este trabajo es también el cierre de una etapa que sin su apoyo no hubiera sido posible.

A mis tutores Marcos y Alberto, les agradezco la oportunidad de realizar este trabajo, su predisposición y el tiempo dedicado.

A mi familia, pareja y amigos, les agradezco el apoyo incondicional, los consejos y la paciencia.

Por último, un agradecimiento especial a Lucía y Cecilia, quienes me ayudaron con la corrección de este trabajo.

Resumen

En este proyecto realizamos un estudio del lenguaje LiquidHaskell, una extensión de Haskell que permite agregar lógica de predicados a las funciones y tipos a través de un SMT-Solver. Inicialmente, realizaremos una breve introducción para luego estudiar tres casos de uso de el mismo. En el desarrollo de estos, podremos ver las diferentes formas de utilizar el lenguaje, así como sus fortalezas y debilidades.

Índice general

1. Introducción **1**

2. Preliminares **3**

2.1. LiquidHaskell 3
2.2. Lógica de refinado 3
2.2.1. SMT-Solver 4
2.2.2. Notación 4
2.2.3. Banderas 4
2.2.4. Preludio 5
2.2.5. Archivos generados 5
2.3. Agregar invariantes a funciones y tipos 5
2.3.1. Funciones 5
2.3.2. Funciones sin interpretar 7
2.3.3. Terminación 8
2.3.4. Tipos 9
2.3.5. GADT's 9
2.4. Parámetros 10
2.5. Promover funciones de Haskell a lógica de predicados 11
2.5.1. Inline 11
2.5.2. Measures 12
2.5.3. Reflection 15
2.5.4. Razonamiento ecuacional 16
2.5.5. Proof by logical evaluation 18

3. Implementación de Árboles Binarios de Búsqueda **21**

3.1. Implementación externalista 21
3.1.1. Insert 23
3.1.2. Delete 24
3.2. Implementación internalista 27
3.2.1. Insert 27
3.2.2. Delete 28
3.3. Comparación entre las implementaciones internalista y externalista 28

4. Implementación de AVL **31**
4.1. Modificaciones estructurales . . . 31
4.2. Peso . . . 33
4.2.1. Insert . . . 34
4.3. Balanceo . . . 35
4.4. Rotaciones . . . 37
4.4.1. Rotaciones simples . . . 37
4.4.2. Rotaciones dobles . . . 38
4.5. Delete . . . 41
4.5.1. Merge . . . 42
4.5.2. Comparación entre implementaciones . . . 44

5. Implementación de BraunTrees **45**
5.1. Estructura . . . 46
5.1.1. Subtipos . . . 47
5.2. Operaciones Básicas . . . 49
5.2.1. Noción de paridad . . . 50
5.2.2. Búsqueda . . . 50
5.2.3. Actualizar . . . 51
5.2.4. Largo . . . 52
5.2.5. Transformación a lista . . . 52
5.2.6. Correctitud funcional . . . 54
5.3. Arreglos flexibles . . . 57
5.3.1. Agregar al inicio . . . 57
5.3.2. Borrar el primer elemento . . . 58
5.3.3. Agregar al final . . . 59
5.3.4. Borrar el último elemento . . . 59
5.4. Operaciones eficientes . . . 60
5.4.1. Tamaño . . . 60
5.4.2. Altura . . . 61
5.4.3. Generar un arreglo de largo n . . . 61
5.4.4. Transformación de una lista a un arreglo . . . 62

6. Conclusiones **69**
Referencias . . . 71

A. Prueba de equivalencia función isABB **73**

Capítulo 1

Introducción

En la actualidad, los programas limitan el dominio de sus funciones a través de los tipos; estos nos brindan conjuntos de elementos de los cuales los valores de la función pueden tomar. Sin embargo, las restricciones en tiempo de compilación que podemos aplicar sobre ellos son extremadamente limitadas, por no decir nulas. Es así que, usualmente, estas restricciones las realizamos en tiempo de ejecución a través de estructuras condicionales.

En este proyecto de grado estudiaremos la herramienta LiquidHaskell, un plugin de GHC que nos permite extender el lenguaje y agregar restricciones en tiempo de compilación en forma de lógica de predicados.

En el segundo capítulo, introduciremos la herramienta, proporcionando una descripción detallada de sus funcionalidades, cómo agregar restricciones a los tipos y las funciones a través del uso de operaciones booleanas simples. Luego, veremos cómo extender esta lógica para poder utilizar las funciones creadas por nosotros y no solo los operadores. Continuaremos analizando cómo razonar con esa lógica para demostrar propiedades sobre las funciones. Por último, una vez introducida la herramienta, profundizaremos en su estudio a través de la implementación de tres estructuras de datos distintas, las cuales expondremos en los siguientes capítulos.

En el tercer capítulo, presentaremos la implementación de los árboles binarios de búsqueda, en la cual utilizaremos LiquidHaskell para hacer cumplir la condición de orden que poseen los mismos. Además, esta estructura fue implementada con dos enfoques: uno externalista y otro internalista. Compararemos ambos enfoques y analizaremos las ventajas y desventajas de cada uno.

En el cuarto capítulo, desarrollaremos una versión de los AVL utilizando la lógica de predicados para verificar que la condición de balanceo de los árboles sea respetada en todas sus operaciones. Exploraremos algunas de las limitaciones de la herramienta, principalmente relacionadas con las formas de escribir las condiciones que queremos verificar.

En el quinto capítulo, implementaremos la estructura BraunTree, donde agregamos la noción de paridad a nuestra lógica de predicados para poder realizar de forma segura las operaciones con estos árboles. Además, profundizaremos

en el razonamiento ecuacional demostrando propiedades que cumplen las funciones de la estructura y que de otra forma requerirían una gran cantidad de condicionales. Por último, implementaremos una serie de mejoras en la eficiencia y proporcionaremos una prueba de cómo el resultado de las funciones eficientes es el mismo que el de las iniciales.

En el capítulo sexto, presentaremos las conclusiones de este trabajo y los principales aprendizajes al utilizar la herramienta, así como el trabajo a futuro.

Capítulo 2

Preliminares

En este capítulo se introducen las principales nociones para programar utilizando LiquidHaskell, su funcionamiento y las herramientas que nos provee. A su vez, estudiaremos cómo agregar invariantes a las funciones o tipos y el proceso por el cual LiquidHaskell las evalúa. También profundizaremos en la noción de lógica de refinamiento, cómo promover funciones de Haskell y que beneficios conlleva. Por último, detallaremos cómo utilizarlas para deducir predicados lógicos a través del razonamiento ecuacional.

2.1. LiquidHaskell

LiquidHaskell (LH) es una extensión del lenguaje Haskell que nos permite agregar propiedades a los tipos o a las funciones de un programa en tiempo de compilación. Además de esto, LH nos permite demostrar predicados lógicos a través de la utilización de deducción ecuacional.

Técnicamente, LH es un plugin del Glasgow Haskell Compiler (GHC) diseñado para aprovechar el gran avance realizado en los probadores de teoremas, los cuales se utilizan para chequear en tiempo de compilación si los invariantes agregados se cumplen. Mediante las técnicas de deducción automática de los probadores de teoremas, LH le permite al usuario agregar las propiedades sin la necesidad de escribir una prueba formal de ellas.

Los lenguajes de tipos dependientes, como Agda (Norell, 2007) o Coq (Bertot y Castéran, 2004) son generalmente comparados con LH. A diferencia de los lenguajes mencionados anteriormente, la automatización de las pruebas mediante el uso de un probador de teoremas nos permite escribir menos líneas de código para implementar lo mismo (Vazou, Rondon, y Jhala, 2013).

2.2. Lógica de refinado

Analizando cómo LH lleva las propiedades escritas en código a invariantes que se chequean en tiempo de compilación, podemos detallar el siguiente pro-

cedimiento. Al momento de compilar un archivo, GHC llama una función al final de la fase de *typechecking*, la cual se encarga de interpretar las anotaciones realizadas en el código Haskell, para luego traducirlo a un lenguaje que pueda ser utilizado por un probador de teoremas. Esto determinará si el código es lógicamente correcto o no.
De esta forma, un proyecto en LH siempre consta de dos códigos:

1. El código fuente escrito en Haskell y anotado para poder agregar invariantes
2. El código utilizado como lógica para deducir que el programa en Haskell cumple las propiedades requeridas.

Es así que denominaremos lógica de refinado como: el conjunto de definiciones, valores y predicados, utilizados por el probador de teoremas para chequear la satisfacibilidad lógica del código. A lo largo de este proyecto se utilizó Z3, el cual es el probador de teoremas recomendado por los autores de LH (Jhala, 2020a), no obstante es posible utilizar CVC4 o MathSat como alternativas.

2.2.1. SMT-Solver

Los *SMT-Solvers* son programas que tienen como propósito intentar determinar si una formula matemática puede ser satisfecha o no. Estos programas son utilizados para probar teoremas de forma automática y es por eso que nos referiremos ocasionalmente a ellos como probadores de teoremas. Cabe destacar que no son los únicos programas que realizan esta función. Lo que los diferencia es que utilizan una serie de tácticas particulares para deducir si la formula se cumple, no se cumple o no se puede determinar.

2.2.2. Notación

LH utiliza anotaciones en código para indicar las propiedades que se deben chequear. Es así que todos los invariantes agregados deben ser escritos dentro de un bloque que comience y termine de la siguiente manera: `{-@ código @-}`. Cada vez que abramos un par de corchetes { } estaremos hablando de un conjunto y será necesario indicarle un tipo { `Int` } . Para hablar de los elementos de dicho conjunto, debemos asignarle un nombre a la representación de los elementos. Esto lo hacemos a través de los dos puntos `n :` { `Int` }. Por último si queremos que el conjunto cumpla ciertas propiedades lógicas, bastará simplemente con escribirlas a la derecha del tipo separado por una barra. $n : \{Int \ \mid \ n > 0\}$ En este caso estamos hablando del conjunto que posee todos los enteros mayores a 0.

2.2.3. Banderas

LH posee un conjunto de funcionalidades por defecto, pero ciertas características pueden ser habilitadas o cambiadas mediante la utilización de banderas

(Jhala, 2020b). Si queremos agregar, restringir o ignorar ciertas funcionalidades o reglas durante el *typechecking* debemos de escribir en el inicio del archivo la siguiente línea `{-@ LIQUID "--bandera"@-}`.

2.2.4. Preludio

LH provee su propia versión del Preludio de Haskell, en él nos encontraremos las mismas funciones y tipos que en la biblioteca original, pero con algunas propiedades ya anotadas. Es así que las funciones del Preludio ya tienen algunas invariantes por defecto y podremos decidir si utilizarlas o reescribirlas.

A su vez, el preludio define funciones que permiten al compilador unificar ciertas transformaciones que realiza el mismo a la lógica de predicados.

2.2.5. Archivos generados

Una vez que uno compila utilizando LH, veremos cómo GHC comienza a procesar archivo por archivo las anotaciones y predicados que especificamos. Cada vez que un archivo se analiza, el compilador serializa el resultado en una serie de archivos que contienen distintos tipos de información que nos puede ser de utilidad. Los archivos generados son:

- Archivos de instrucciones: Estos archivos llevan la extensión .smt2 y contienen las instrucciones .smt2 utilizadas por SMT-Solver para chequear las condiciones y el resultado.

- Archivos de errores: Contienen la información del error que detectó LH. Esta información se serializa en distintos formatos y puede ser accedida por el usuario para ver los errores presentados de forma más amigable.

2.3. Agregar invariantes a funciones y tipos

La principal utilidad que nos provee LH es el agregado de propiedades a las funciones y los tipos. En esta sección introduciremos las principales formas de agregar propiedades lógicas a un programa en Haskell.

2.3.1. Funciones

Dentro de las funciones, LH nos permite definir propiedades sobre su dominio o codominio. Otra forma de interpretar esto es el agregar pre y postcondiciones. La manera que tenemos para indicarle a LH que estamos agregando dichas condiciones a la función es mediante la anotación de la firma de la misma. Supongamos que queremos escribir la función división de forma segura; para esto, 0 no puede ser un valor del divisor. La forma más usual de hacer esto en Haskell es a través de la monada `Maybe`:

```haskell
safe_div :: (Fractional a, Eq a) => a -> a -> Maybe a
safe_div _ 0 = Nothing
safe_div x y = Just (x / y)
```

Esto resuelve nuestro problema, pero a partir del uso de `safe_div` será necesario trabajar con la monada `Maybe`. Al utilizar monadas, estamos agregando condiciones que se chequean en tiempo de ejecución. Si bien esto hace que la división sea segura, es posible ejecutar `safe_div n 0`.

LH nos brinda una solución que nos permite no tener que preocuparnos de esto. Agregando la lógica mencionada anteriormente como una precondición (Jhala, 2013) de la función, podemos obtener una implementación segura de la división, presentada en el código a continuación:

```haskell
{-@ safe_div :: Int -> d : { Int | d != 0 } -> Int @-}
safe_div :: Int -> Int -> Int
safe_div x y = div x y
```

Como podemos ver, sustituimos el tipo del divisor por un conjunto de elementos que no incluya el cero. Además, al ser un chequeo en tiempo de compilación, nos aseguramos que nunca se pueda llamar a la función con el parámetro del divisor con valor cero. Sumado a esto, ya no tendremos que estar preguntando si la operación se realizó de forma correcta o no, el simple hecho de llamarla nos dará un resultado.

Similar al caso anterior, podremos establecer una postcondición como en el siguiente ejemplo, en el cual definimos que el resultado de la suma de enteros positivos es positivo.

```haskell
{-@ nsum :: a : { Int | a > 0 } -> b: { Int | b > 0 } ->
  c : { Int | c > 0 }@-}
nsum :: Int -> Int -> Int
nsum a b = a + b
```

LH verificará que el cuerpo de la función efectivamente sea mayor a cero, por lo que si eliminamos las precondiciones al momento de compilar, obtendríamos un error de tipos.

Entonces, dada la especificación de `safe_div` podremos realizar la división de forma segura y en caso de que el compilador no pueda chequear que el divisor no es 0 LH nos advertirá que debemos demostrar que el mismo lo es. A continuación podemos ver algunos ejemplos:

```haskell
safe_div 4 2 -- retorna 2
safe_div x 2 -- retorna div x 2
safe_div 1 (nsum x y)
safe_div x y -- falla y nos solicita una prueba de qué y != 0
```

En casos como el tercero podemos ver que LH deduce que el resultado de la suma será distinto de 0, sin embargo, para el cuarto será necesario proporcionar una demostración. Esto puede ser solucionado proporcionando un simple chequeo:

```
division :: Int -> Int -> Int
division x y
  | y != 0 = safe_div x y
  | otherwise = -- ejecutar otro caso
```

Si observamos no fue necesario indicarle a `safe_div` que `y` es distinto de 0 a través de notaciones, sino que LH puede deducir que cumple la condición a partir del código Haskell.

Esto tiene sus limitaciones, supongamos ahora que queremos saber si el resultado de la división es 0, lo más común sería escribir la función resultIs0 de la siguiente manera:

```
resultIs0 :: Int -> Int -> Bool
resultIs0 x y = (y /= 0) && (safe_div x y == 0)
```

Sin embargo, LH no es capaz de deducir que `y` es distinto que 0 (no es capaz de evaluar el circuito corto únicamente). Eso se debe a que cada condición debe ser verdadera bajo el contexto en el que se encuentra evaluada. Como y distinto de 0 no es parte del contexto de la condición, esta falla. La forma de arreglar esto es mediante la adición de un condicional.

```
resultIs0 :: Int -> Int -> Bool
resultIs0 x y
  | (y /= 0) = (safe_div x y == 0)
  | otherwise = False
```

De esta forma nos aseguramos de que la evaluación de `safe_div` tiene dentro de su contexto la propiedad y es distinta de 0. Esto se debe a cómo LH interpreta las funciones o, en este caso, cómo LH no interpreta las funciones, sino que para la herramienta son cajas negras las que, dependiendo de una entrada, resultan en una salida.

2.3.2. Funciones sin interpretar

Visto el ejemplo anterior, nos interesa profundizar en cómo LH une el código Haskell con el SMT-solver. El SMT-solver trata a las funciones provenientes de Haskell como funciones sin interpretar (Jhala, Seidel, y Vazou, 2023), esto significa que para él estas son cajas negras que cumplen con el axioma de congruencia el cual establece que la evaluación de una función con el mismo argumento retorna el mismo resultado.

Es así que el SMT-solver nos permite proveer poca información a las funciones siempre y cuando el axioma de congruencia se respete. Sin embargo, esto imposibilita evaluar el circuito corto porque dado la entrada cada predicado se evalúa de forma independiente con el contexto que tiene a disposición, no se agrega la información necesaria al contexto y esto puede terminar en errores que codificados en Haskell estén bien, pero que LH no posee la capacidad de validarlos.

2.3.3. Terminación

Si bien Haskell es un lenguaje que se basa en la evaluación perezosa, la necesidad de una correcta verificación de los tipos determina que LH requiera que las funciones tengan terminación. Esto se debe a que una función que devuelve un int, pero es infinita, no devuelve un int, por lo que LH la considera como incorrecta. Para que la utilización de LH no derive en escribir código Haskell de forma estricta, el mismo corrobora que el conjunto de elementos posibles para el primer atributo decrezca (Vazou, Seidel, Jhala, Vytiniotis, y Peyton-Jones, 2014), esto puede verificarse de varias maneras en el caso de las listas es a través de su longitud en el caso de los enteros por su valor.

A continuación podemos ver un ejemplo de esto con la función de Fibonacci sobre los enteros:

```haskell
{-@ fib :: i:Int -> Int  @-}
fib :: Int -> Int
fib i | i == 0    = 0
      | i == 1    = 1
      | otherwise = fib (i-1) + fib (i-2)
```

En este caso, LH marcará un error porque no puede definir si la función fib terminará o no, por lo que existe la posibilidad de que la función se vaya del rango de los enteros. Sin embargo, sería fácil definir como precondición los enteros positivos, solucionando el problema. El ejemplo anterior fue fácil de solucionar; en algunos casos esto no es tan trivial, por lo que pueden surgir complicaciones adicionales. Cuando nos encontramos frente a estos casos, LH nos provee las siguientes opciones:

1. Cambiar el dominio de la función para que este dentro de los naturales
2. Indicar que la función no termina mediante la anotación lazy: `{-@ lazy foo @-}`
3. Desactivar el chequeo de terminación mediante la bandera `{-@ LIQUID -no-termination"@-}`

Sumado a esto, si la función es de múltiples parámetros, es posible indicar sobre cuál está siendo ejecutada. La forma de notar esto es mediante la adición del operador / seguido del nombre del parámetro entre paréntesis rectos al tipo de retorno de la función. A continuación podemos ver un ejemplo:

```
{-@ repeat' :: i : Int -> n : Nat -> [Int] / [n] @-}
repeat' :: Int -> Int -> [Int]
repeat' i 0 = []
repeat' i n = i : (repeat' i (n - 1))
```

2.3.4. Tipos

Agregar invariantes a las funciones es de gran utilidad, pero si todo el tiempo estamos agregando las mismas pre y postcondiciones a las funciones, el código se puede volver muy repetitivo. Definir conjuntos que cumplan con esas condiciones y puedan ser utilizados en las funciones facilita la escritura y legibilidad, es por eso que LH nos permite agregar invariantes a los tipos. A diferencia de las funciones, estas no tienen una firma asociada, sino que, los anotaremos mediante la directiva `type`. De esta forma, LH nos permitirá crear subconjuntos dentro de los tipos donde se cumplan ciertas propiedades lógicas que nos pueden interesar.

Uno de los ejemplos más básicos de tipos que podemos redefinir son los enteros `Int`, si por ejemplo queremos contar con el subconjunto de los Naturales, los Naturales positivos o los enteros sin el cero, podemos definirlos de la siguiente forma:

```
{-@ type Nat = { x : Int | x >= 0 } @-}
{-@ type Pos = { x : Int | x > 0 } @-}
{-@ type NonZero = { x : Int | x != 0 } @-}
```

Estas definiciones podemos utilizarlas luego en nuestras funciones para ahorrarnos escribir la misma condición todo el tiempo:

```
{-@ safe_div' :: Int -> NonZero -> Int @-}
safe_div' :: Int -> Int -> Int
safe_div' x y = div x y
```

Como podemos ver, si bien en la anotación de LH tenemos el subconjunto de los naturales sin el cero, en el código Haskell esta restricción no existe. Esto es un punto importante a notar. LH realiza los chequeos en tiempo de compilación, pero nada nos impide, una vez ejecutado el programa usar funciones aisladas de forma incorrecta. Si nosotros cargamos en ghci la biblioteca que contenga `safe_div'`, ghc compila y verifica que satisface todas nuestras propiedades, pero una vez terminado este proceso nada nos impide llamar a `safe_div' 1 0`, lo cual causaría un error en tiempo de ejecución.

2.3.5. GADT's

Además de poder definir subconjuntos de tipos, LH nos permite agregar lógica de predicados a los GADT's. Supongamos que queremos construir el GADT que representa a una lista ordenada. Para eso utilizaremos el siguiente GADT:

```
data SortedList a where
    Cons :: (Ord a) => a -> SortedList a -> SortedList a
    Empty :: (Ord a) => SortedList a
```

Bastará simplemente con redefinirlo dentro de anotaciones con la lógica que nosotros queremos que la estructura represente. En este caso, al ser una lista ordenada, le indicaremos que el conjunto de elementos admitidos en el siguiente constructor tiene que ser mayor o igual al previo.

```
{-@
data SortedList a where
    Cons :: (Ord a) => v : a ->
     SortedList { vv : a | v <= vv } -> SortedList a
    Empty :: (Ord a) => SortedList a
@-}

t1 = Cons 1 (Cons 2 (Cons 3 Empty))

t2 = Cons 3 (Cons 2 (Cons 3 Empty))
```

En este caso, LH nos indicará que el elemento t1 es correcto, mientras que el t2 tendrá un error porque el valor inicial de la lista es mayor al del siguiente 3 > 2. A diferencia de la directiva type al redefinir la estructura, no es necesario especificarle a LH que el tipo de t1 es SortedList.

2.4. Parámetros

LH permite diferenciar los parámetros en código anotado entre tipos y valores; esto es indicado por el tipo de carácter utilizado al momento de nombrarlos. Si queremos utilizar valores, el nombre del parámetro tiene que estar todo en mayúscula. Por el contrario, si queremos utilizar tipos, el nombre tiene que estar en minúscula.

Supongamos que queremos crear el subtipo de la lista de largo conocido n. El preludio de LH nos provee la función *len* que nos permite saber el largo de una lista. Es así que basta únicamente definir el tipo de la siguiente forma:

```
  {-@ type ListN a N = l: { [a] | len l == N } @-}
```

En este caso `a` es un tipo mientras `N` es un valor entero que define el largo de la lista.

2.5. Promover funciones de Haskell a lógica de predicados

Como pudimos ver en las secciones anteriores, es posible agregar invariantes tanto a tipos como a funciones, pero si observamos con cuidado, esto se realizó únicamente mediante la utilización de operadores *booleanos* o funciones que nos provee el preludio. Si nosotros quisiéramos usar la función `safe_div` dentro de la lógica de refinamiento, no sería posible, ya que LH no tiene forma de interpretarla.

Si bien, es verdad que en un principio agregar predicados lógicos es de utilidad, nuestro interés es lograr invariantes más complejos, a través de la creación de funciones que puedan ser utilizadas dentro de la lógica de refinado. Es así que LH nos permite promover de varias formas funciones escritas en Haskell para poder utilizarlas luego al momento de escribir predicados.

2.5.1. Inline

Una de las formas de promover funciones y predicados a la lógica de refinado es a través de la instrucción `inline`. La misma promueve una función de Haskell a la lógica de refinamiento que cumpla con las siguientes condiciones:

- Todos los componentes utilizados en la función deben estar disponibles en la lógica de refinamiento
- No puede ser recursiva

Es así, que ahora podemos renombrar ciertas condiciones que nosotros escribíamos de forma literal, como puede ser el chequeo de que un entero sea par. A continuación podemos ver la implementación de la misma:

```
{-@ inline even @-}
even :: Int -> Bool
even x = x `mod` 2 == 0
```

Una vez definida como `inline` podremos utilizar esta función para definir el conjunto de los números pares e impares, lo que no es posible para funciones que no poseen la notación `inline`.

```
{-@ type Even = { x : Int | even x } @-}
{-@ type Odd = { x : Int | not (even x) } @-}
```

Si bien la directiva `inline` es útil porque nos permite agregar funciones a la lógica de refinado, las mismas son de una complejidad baja. Se utilizan principalmente para crear aliases de condiciones, ya que si no deberíamos escribirlas de forma literal, lo cual se puede tornar engorroso. El no permitir recursión es un gran problema, porque no será posible definir, por ejemplo, que todos los elementos de una lista de enteros sean pares.

2.5.2. Measures

Una *measure* es otra de las formas que nos provee LH de promover una función a la lógica de refinado. Para esto nuestra función debe cumplir las siguientes tres condiciones:

- Debe ser definida de forma inductiva
- Debe tener un único argumento de entrada
- Debe ser decreciente

Una vez que nuestra función cumple estas condiciones y nosotros utilizamos la directiva `measure` en la firma de la función `{-@ measure foo @-}` LH adjuntara la función sin interpretar al constructor del primer parámetro (Vazou y cols., 2014)

Como la función es inductiva, decreciente y de un único argumento, será posible adjuntarla sin interpretar ya que la misma podrá ser procesada por el *SMT-solver* en caso de ser necesario, porque nunca se quedará sin terminar. Volviendo al ejemplo anterior en el que los elementos de una lista deben ser todos pares, podemos definir una función de la siguiente manera para chequear esta propiedad:

```
{-@ measure allEven @-}
allEven :: [Int] -> Bool
allEven []= True
allEven (x:xs) = even x && allEven xs
```

Luego con esta `measure` definida podremos utilizarla para definir condiciones de elementos que necesitemos cumplan esta propiedad. Un ejemplo de ese uso podemos verlo a continuación:

```

{-@ type EvenList = xs: { [Int] | allEven xs } @-}

{-@ t1 :: EvenList @-}
t1 :: [Int]
t1 = [2,4,6,8]
```

Gracias a las `measures` fuimos capaces de agregar lógica que no sería posible si quisiéramos utilizar `inline`. En este caso, agregamos el ejemplo t1 que nos permiten chequear a modo de test que el invariante se chequea de forma correcta.

Sin embargo, si queremos chequear que el inverso no se cumple, es decir, que si agregamos un elemento impar a nuestra lista no va a compilar. Para estos casos, LH nos provee la directiva `fail` una forma de indicarle que la falla es el caso esperado y así poder probar que el invariante se cumple en los casos correctos e incorrectos. A continuación, podemos ver a `t2` y `t3` ejemplos de lo mencionado anteriormente:

```haskell
{-@ t2 :: EvenList@-}
{-@ fail t2 @-}
t2 :: [Int]
t2 = [3]

{-@ t3 :: EvenList@-}
{-@ fail t3 @-}
t3 :: [Int]
t3 = [2,4,3,6,8]
```

Esta directiva es de mucha utilidad porque cuando promovemos una función a una `measure` no siempre tenemos en cuenta la totalidad de condiciones que tiene la estructura con la que estamos trabajando. Como resultado podemos anular condiciones sin que esta sea nuestra intención.

Para familiarizarnos con el uso de las `measures` utilizaremos la función largo de las listas. Supongamos que queremos construir la estructura vector. Una forma de hacerlo es a través de una estructura similar a la utilizada en la Sección 2.3.5 para representar listas ordenadas y agregar un elemento con el largo del vector.

```haskell
data Vector a where
    Cons :: a -> Int -> Vector a -> Vector a
    Empty :: Vector a
```

Queremos que el elemento entero dicte el largo del mismo, para eso necesitaremos una función largo que, dada la lista, devuelva la cantidad de elementos que tiene la misma:

```haskell
{-@ measure length @-}
length :: Vector a -> Int
length (Cons _ _ next) = 1 + length next
length (Empty) = 0
```

Al promover la función `length` a la lógica de predicados, LH la adjunta al constructor de la estructura; esto en la lógica de refinamiento tendrá la siguiente forma:

```haskell
 data Vector a where
  Cons :: a -> n : Int -> sl : Vector a ->
    { res : Vector a | length res = 1 + length sl }
  Empty :: { sl : Vector a | length sl  = 0 }
```

Como la función se promovió, ahora podemos usarla para limitar los vectores que construimos, de la misma forma en la que condicionábamos al GADT en la Sección 2.3.5. A continuación podemos ver cómo limitamos los vectores que pueden ser limitados por el parámetro largo que pertenece al GADT.

```
{-@
data Vector a where
    Cons :: a -> l : Int ->
        next : { Vector a | l == length next + 1 }-> Vector a
    Empty :: Vector a
    -@}
```

Un ejemplo de un vector podría ser el siguiente:

```
test1 :: Vector Int
test1 = Cons 1 4 (Cons 2 3 (Cons 3 2 (Cons 4 1 Empty)))

{-@ fail test2 @-}
test2 :: Vector Int
test2 = Cons 1 5 (Cons 2 3 (Cons 3 2 (Cons 4 1 Empty)))
```

Pattern matching

Al definirnos las funciones como `measures` y adjuntar la función al constructor, ganamos la posibilidad de hacer *pattern matching* sin la necesidad de definir todos los casos. Esto quiere decir que, si nosotros agregamos condiciones a las funciones que filtren alguno de los constructores, no será necesario declararlos en la misma. Por ejemplo, si nosotros definimos la función notEmpty

```
{-@ measure notEmpty @-}
notEmpty :: Vector a -> Bool
notEmpty Empty = False
notEmpty _ = True
```

Luego podemos utilizarla para crear funciones que accedan al vector no vacío sin la necesidad de declarar el caso:

```
{-@ value :: t :{ Vector a | notEmpty t} -> a @-}
value :: (Ord a) => Vector a -> a
value (Cons v _ _) = v
```

Esto es de mucha utilidad porque LH puede deducir las precondiciones sin problema y podemos escribir menos código, asumiendo que nunca se podrá llamar a la función `value` con un Vector vacío.

Measures sobre tipos específicos

En la sección anterior vimos dos ejemplos de `measures` distintas, la primera, sobre una lista de enteros, la segunda sobre la estructura de un Vector genérica. Ahora veamos que pasa si plantemos la función `allEvenV` con el mismo comportamiento de `allEven`, pero que opere sobre los vectores:

```haskell
{-@ measure allEvenV @-}
{-@ allEvenV :: Vector Int -> Bool @-}
allEvenV :: Vector Int -> Bool
allEvenV Empty = True
allEvenV (Cons v _ next) = even v && allEvenV next
```

Esto, sin embargo, retornará un error de especificación. Este error se debe a cómo LH define las `measures`. Como vimos en la sección anterior, la función se adjunta al constructor del tipo, en este caso `Vector Int`. El problema se presenta porque nuestro vector es definido como un tipo paramétrico `Vector a` y luego instanciado a `Int`. LH adjunta por defecto la función al constructor genérico de las `measures`, pero en nuestro ejemplo, la función `even` requiere que los tipos sean estrictamente `Int`. Debido a esto, LH termina dando un error de especificación. Para que no adjunte funciones específicas sobre tipos genéricos, será necesario indicarlo mediante el uso de la bandera `--prune-unsorted`. Cabe destacar que para el ejemplo de las listas pares fue necesario utilizar esta funcionalidad.

2.5.3. Reflection

Si bien las `measures` resultan de mucha utilidad, tienen limitaciones muy fuertes: la restricción de utilizar únicamente un solo parámetro, la incapacidad de utilizar funciones que no sean decrecientes y la agregación de funciones a los constructores de los tipos de una función. Es de interés poder usar el SMT-solver para poder chequear funciones que no cumplan con las limitaciones que poseen las `measures`.

Como alternativa podemos usar la directiva `reflect` utilizada para llevar funciones escritas en Haskell a nuestra lógica de refinado (Vazou y cols., 2017). Para poder activar la directiva reflect es necesario indicarlo mediante la bandera `--reflection`.

El proceso de reflejar una función consiste en que, durante la compilación, se realizan los siguientes tres pasos sobre la misma:

- Crear una función sin interpretar que solo satisface el axioma de congruencia.
- Se refleja la definición de la función en un tipo refinado
- Con la función reflejada es posible hacer *unfolding* de la misma cada vez que se utiliza en la lógica de refinado.

Es así que si vamos al ejemplo utilizado con las measures:

```haskell
{-@ reflect length @-}
length :: Tree a -> Int
length (Cons _ _ next) = 1 + length next
length (Empty) = 0
```

El código generado al pasar a la lógica de refinamiento será el siguiente:

```
length :: t: Tree -> { n: Nat | n = length t && lengthP t }

-- Donde lengthP es un alias del refinamiento derivado de la
-- función
lengthP t = t == Empty => length t = 0 &&
         t == (Cons v l next)  => length t = 1 + length next
```

A diferencia de las `measures` las funciones que utiliza la directiva `reflect` no proporcionan ningún tipo de refinamiento sobre los constructores. Es así que si intentamos usar el ejemplo del Vector con largo 4 con la nueva definición, la compilación fallará porque LH no hace el *unfolding* de la función y no tienen ninguna información del largo del Vector. Esto presenta un problema, ya que si bien las funciones reflejadas son más flexibles que las `measures` estas no poseen una forma directa de automatizar las pruebas de las pre y postcondiciones en las que las usamos.

En las siguientes secciones veremos algunos agregados que nos permitirán hacer que las funciones reflejadas posean características similares a las `measures`.

2.5.4. Razonamiento ecuacional

LH provee una biblioteca para realizar pruebas a través de razonamiento ecuacional(Vazou, Breitner, Kunkel, Van Horn, y Hutton, 2018), con esta biblioteca podremos utilizar todas las funciones promovidas a la lógica de refinado para demostrar teoremas utilizando este tipo de razonamiento.

La biblioteca introduce las siguientes directivas para realizar pruebas:

- `Proof` un alias de () utilizado principalmente para indicar que la función que estamos construyendo es una prueba
- `trivial` indica que la prueba es trivial y puede ser resuelta automáticamente por el SMT solver.
- `QED` es un tipo que actúa como certificación de que la deducción es lo suficientemente fuerte como para que el SMT solver pruebe el teorema.
- `Admit` indica que una prueba no está finalizada.
- `(***)` operador utilizado para indicar la finalización de una prueba es el antecesor a `QED` o `Admit` los cuales indican si el *SMT-solver* puede o no resolver la derivación.
- `(===)` operador que indica igualdad entre las ecuaciones.
- `(=<=)` operador que indica la relación de menor entre dos estructuras de la clase `Ord`.

- `(=>=)` operador que indica la relación de mayor entre las estructuras de la clase `Ord`.

- `?` Nos permite agregar nuevos elementos para probar nuestro teorema.

Una vez definidos estos operadores, podemos pasar a demostrar propiedades sobre las estructuras que teníamos anteriormente. Supongamos que nos interesa agregar una función que pregunte si el vector es vacío, esto es posible mediante la siguiente implementación.

```
isEmpty :: Vector a -> Bool
isEmpty Empty = True
isEmpty _ = False
{-@ reflect isEmpty @-}
```

Ahora queremos demostrar que, dado un vector, si este es vacío, entonces el mismo es igual a *Empty*, sería tan solo definirnos la propiedad y la demostración es muy fácil:

```
{-@ isEmptyThenEmpty :: t : { Vector a | isEmpty t } ->
  { t == Empty }@-}
isEmptyThenEmpty :: Vector a -> Proof
isEmptyThenEmpty Empty = trivial *** QED
isEmptyThenEmpty t = isEmpty t == False *** QED
```

Como podemos ver, a diferencia de las `measures` las funciones `reflect` no pueden filtrar los constructores. Para poder satisfacer al *solver* tendremos que proveer una demostración para cada caso. El primero es trivial, por lo que utilizamos la directiva del mismo nombre. El segundo, nos encontramos frente a un absurdo, `isEmpty` nunca podría ser verdad porque el vector nunca es vacío, por lo que negamos la precondición.

Supongamos que, además de la propiedad anterior, nos interesa demostrar que si insertamos un elemento en un vector, el resultado tendrá altura 1. Para eso definiremos la función insert de la siguiente manera:

```
{-@ reflect insert @-}
insert :: (Ord a) => Vector a -> a -> Vector a
insert (Empty) v =  Cons v 1 Empty
insert t@(Cons v l next) v2 = Cons v2 (l + 1) t
```

Esta es una simple implementación de la función `insert`, pero esta función sería imposible promoverla como `measure` porque no tiene un solo parámetro de entrada. Una vez definida la función, podemos demostrar la propiedad de la siguiente manera:

```haskell
{-@ insertLength :: t :  Vector a -> v: a ->
    { length (insert t v) == 1 + length t } @-}
insertLength ::(Ord a) => Vector a -> a -> Proof
insertLength Empty v =
        length (insert Empty v) == 1 + length Empty
    === length (Cons v 1 Empty) == 1
    === 1 + length Empty == 1
    === 1 == 1 *** QED
insertLength vec@(Cons v l next) v2 =
        length (insert vec v2) == 1 + length vec
    ===  length (Cons v2 (l + 1) vec) == 1 + length vec
    === 1 + length vec == 1 + length vec
    *** QED
```

Como podemos ver utilizamos *pattern matching* nuevamente y en ambos casos desarrollamos a través de equivalencias. En cada paso remplazamos las funciones `insert` o `length` por sus evaluaciones. Una vez llegamos a un punto donde ambos lados de la igualdad valen lo mismo el teorema queda demostrado.

A diferencia de Coq, Agda y otros lenguajes, LH no tiene ningún tipo de noción de computación, normalización, canonicidad o reescritura. Podemos simularla a partir del chequeo de equivalencias entre las definiciones de funciones. (Vazou y cols., 2017)

2.5.5. Proof by logical evaluation

Las pruebas que vimos en la sección anterior son fáciles de demostrar porque sus enunciados son bastante simples. A medida que la complejidad de los enunciados aumenta, crecerá el tamaño de las demostraciones y el tiempo que nos insumen. Es así que para simplificar esto, LH nos provee con la posibilidad de utilizar la técnica de *Proof by logical evaluation (PLE)* (Vazou y cols., 2018).

Al agregar esta funcionalidad es posible que las pruebas realizadas a través de razonamiento ecuacional se deduzcan de forma automática. Esta funcionalidad agrega los siguientes pasos al proceso de chequeo de la lógica de refinamiento:

1. Se transforma toda función en su forma reflejada
2. Hace el *unfolding* de las funciones reflejadas.
3. Repite los pasos anteriores hasta que llegue a un punto fijo.

Agregar estas fases al proceso de refinamiento es de gran utilidad. Para indicarle a LH que queremos utilizar PLE sera posible hacerlo mediante una de las dos siguientes formas:

1. **Local:** Si queremos activar la instrucción ple de forma local debemos indicarlo mediante el uso de la bandera {`-@ LIQUID -ple-local"@-`}. Para que los elementos que queramos sean procesados utilizando ple debemos

indicarlo mediante la directiva `ple` y el nombre de la función. Unicamente serán procesadas las funciones que tienen la anotación ple dejando así afuera a todos las demás.

2. **Global:** En caso de querer activarlo de forma global bastará con agregar la bandera {`-@ LIQUID -ple"@-`} y todo las funciones presentes en el archivo serán procesadas por este proceso.

Como bien indican los nombres en una solo utilizamos PLE sobre un conjunto de funciones determinadas por nosotros mientras que en el global se realiza el proceso a todas las funciones del archivo. Agregar la funcionalidad PLE no es gratis y en archivos grandes o con muchas condiciones puede generar problemas activarlo de forma global ya que el tiempo de compilación puede llegar a crecer de forma dramática como veremos en el Capitulo 5. Sin embargo, utilizado de forma global permite que los invariantes promovidos por la directiva `reflect` requieran menos cantidad de pruebas asociadas. Si volvemos al ejemplo de la función `length` de la Sección 2.5.3 y utilizamos PLE de forma global podemos utilizar la función de forma similar a una `measure` ya que el compilador realizará el *unfolding* de la misma si la utilizamos en nuestra lógica de refinado.

Veamos ahora los beneficios de PLE en combinación con las pruebas a través del razonamiento ecuacional. Utilizando los ejemplos de la sección anterior, si indicamos de forma local el uso de PLE basta únicamente agregar la directiva con el nombre de la función y llamarla a sí misma en el cuerpo de la prueba para poder demostrar automáticamente el teorema `insertLength`.

```
{-@ ple insertLength2 @-}
{-@ insertLength2 :: (Ord a) => t :  Vector a -> v: a ->
  { length (insert t v) == 1 + length t }@-}
insertLength2 ::(Ord a) => Vector a -> a -> Proof
insertLength2 Empty v = trivial *** QED
insertLength2 vec@(Cons v l next) v2 = insertLength2 next v
```

Como podemos ver la cantidad de código se redujo, lo cual es una de las grandes beneficios de esta herramienta. Las demostraciones por razonamiento ecuacional pueden tornarse extensas y tediosas al momento de implementarlas. Si utilizamos PLE al ser una prueba automática podemos ahórranos el escribir una demostración, sin embargo el tiempo que esta demora será determinado por lo que le lleve al compilador y al SMT-solver resolver la prueba.

Capítulo 3

Implementación de Árboles Binarios de Búsqueda

En esta sección veremos dos implementaciones de Árboles Binarios de Búsqueda (ABB) que nos permite realizar LH, una externalista y otra internalista.

Los ABB se caracterizan por cumplir con la siguiente condición: dado un árbol ABB, los valores pertenecientes a la rama izquierda de un nodo son menores que el valor del nodo padre, análogamente los valores en la rama derecha son mayores al valor del nodo padre. Para establecer un punto de partida, definiremos la estructura de un árbol de la siguiente manera:

```
data Tree a where
  Node :: (Ord a) => a -> Tree a -> Tree a -> Tree a
  Nil :: Tree a
```

3.1. Implementación externalista

Una implementación externalista es aquella en la que la propiedad de los árboles ABB no es parte de la propia construcción del tipo `Tree`, sino que es algo realizado por una función externa. En este caso, `isABB` (de León, 2024a) será la prueba externa encargada de chequear que el árbol cumple con la condición.

Como explicamos en la sección anterior, el árbol deberá cumplir con ciertos criterios de orden. Para que esto sea posible, debemos definir algunas funciones sobre los árboles que nos permitan tener más información de los valores máximos y mínimos de los mismos. Ya que sin esto no será posible comparar el nodo padre con los subárboles.

Es así que definiremos las siguientes funciones: `greatest`, que dado un árbol retorna el valor más grande, y `lowest` que dado un árbol devuelve el valor más chico. Cabe destacar que este tipo de función solo puede ser aplicada a árboles no vacíos, porque si no, no existirían estos valores. Esto se ve reflejado en el siguiente

código como la precondición `notEmpty t` que tiene los parámetros de entrada de ambas funciones. Además, ambas utilizarán la notación `measure` porque luego las utilizaremos para promover a la lógica de refinamiento la condición de que un árbol es ABB.

```haskell
{-@ measure greatest @-}
{-@ greatest :: (Ord a) => t: { Tree a | notEmpty t } ->
  v : a  @-}
greatest :: (Ord a) => Tree a -> a
greatest (Node v l r)
  | notEmpty l && notEmpty r =
    max (max (greatest l) (greatest r)) v
  | notEmpty l = max (greatest l) v
  | notEmpty r = max (greatest r) v
  | otherwise = v

{-@ measure lowest @-}
{-@ lowest :: (Ord a) => t: { Tree a | notEmpty t } ->
  v :  a @-}
lowest :: (Ord a) => Tree a -> a
lowest (Node v l r)
  | notEmpty l && notEmpty r = min (min (lowest l) (lowest r)) v
  | notEmpty l = min (lowest l) v
  | notEmpty r = min (lowest r) v
  | otherwise = v
```

Como podemos ver en la construcción de estas funciones, en ningún momento se asume que la estructura es un ABB, sino que buscan el valor mayor o menor en comparación con todo el árbol. Esto se debe a que por ahora no agregamos ningún tipo de información a la misma. Conforme vayamos agregando condiciones a nuestra estructura, podremos asumir comportamientos dentro de nuestras funciones que sean típicos de los ABB.

Teniendo una forma de obtener el mínimo y el máximo de un árbol, de forma recursiva chequearemos que para todos los nodos el valor del nodo padre sea mayor al máximo del subárbol izquierdo y menor al mínimo del lado derecho. Esto lo podemos ver representado a continuación:

```haskell
{-@ isABB :: (Ord a) => t : Tree a -> Bool @-}
isABB :: Tree a -> Bool
isABB (Nil) = True
isABB  t@(Node v l r)
  | notEmpty l && notEmpty r && leavesABB = v > (greatest l) &&
    v < (lowest r)
  | notEmpty l && leavesABB = v > (greatest l)
  | notEmpty r && leavesABB = v < (lowest r)
  | otherwise = leavesABB
  where
    leavesABB = isABB l && isABB r
{-@ measure isABB @-}
```

Luego definimos los subtipos `ABB` y `ABBNE` para terminar de definir los ABB y los ABB no vacíos. Este segundo subtipo nos servirá para abreviar la condición de que no sean vacíos los árboles.

```haskell
{-@ type ABB a = t: { Tree a | isABB t } @-}
{-@ type ABBNE a = t: { Tree a | isABB t && notEmpty t } @-}
```

3.1.1. Insert

La función `insert`, agregará un elemento nuevo al árbol en caso de que este no esté presente en el mismo. Para esto es necesario plantear una función que primero recorra el árbol de forma recursiva, respetando la condición de ABB. Si durante este recorrido, si nos encontramos el elemento, mantendremos el árbol sin cambios. En caso de llegar a una hoja vacía, se le agrega un nodo con el elemento. A continuación podemos ver la implementación de la función `insert`:

```haskell
insert :: (Ord a) => Tree a -> a -> Tree a
insert (Nil) v = Node v Nil Nil
insert t@(Node v l r) k
    | v > k = Node v (insert l k) r
    | v < k = Node v l (insert r k)
    | otherwise = t
```

Si nosotros le asignamos el subtipo `ABB` a la firma de la función y compilamos LH retornará un error. Esto se debe a que cada vez que llamamos a la función `insert` sobre un subárbol, lo modificamos; por lo tanto, es posible que los valores de mínimo y máximo cambien. Esto LH lo detecta y nos exige que demostremos que la operación realizada no rompe con la condición de los `ABB` Para esto haremos uso de una postcondición sobre la función definida a continuación:

```haskell
{-@ insertPostCondition:: (Ord a) =>
  tf : ABBNE a -> ABB a -> a -> Bool @-}
insertPostCondition :: (Ord a) => Tree a -> Tree a -> a -> Bool
insertPostCondition tf t v
    | notEmpty t =
      (greatest tf == greatest t || greatest tf == v) &&
      (lowest tf == lowest t || lowest tf == v)
    | otherwise = (greatest tf == v) && (lowest tf == v)
{-@ inline insertPostCondition @-}
```

donde el primer parámetro de la función es el árbol resultado de la operación insert y los otros dos parámetros son el árbol antes de la operación y el elemento a insertar.

Por último, modificamos la firma de la función *insert* para que cumpla la especificación:

```haskell
{-@ insert :: t : ABB a -> v : a ->
  tf: { ABBNE a |insertPostCondition tf t v } @-}
```

Lo importante a destacar es que, si bien los elementos máximo y mínimo pueden cambiar, son elementos que conocemos, por lo que es fácil indicarle al compilador el cambio para que pueda decidir si cumple con la condición isABB. Lo que hace esto posible es la inclusión a la función del subtipo ABB; ya que nos otorga información del orden y dominio de cada subárbol. Como todos los subárboles de un ABB son ABB, LH es capaz de mantener las desigualdades y chequear todas las condiciones.

3.1.2. Delete

La función `delete` es una función que busca un elemento en un árbol y lo borra. Al realizar la operación, une las ramas del nodo, cuyo valor fue borrado en caso de que sea necesario. Es así que el código a implementar sin chequear las propiedades de ABB de LH será el siguiente:

```haskell
delete :: (Ord a) => Tree a -> a -> Tree a
delete (Nil) _ = Nil
delete t@(Node v l r) k
    | v > k = Node v (delete l k) r
    | v < k = Node v l (delete r k)
    | otherwise = merge r l
```

Como podemos ver, esto requiere reconstruir el árbol una vez eliminado el elemento. Para esto, si queremos hacerlo con LH la función `delete` deberá poseer una postcondición que permita brindar información del árbol resultante de la operación, ya que si no, no tendrá forma de chequear nuevamente la condición

isABB. Se nos agrega una complejidad en caso de que el elemento eliminado sea el mínimo o máximo, a diferencia de la función anterior no tenemos acceso al posible nuevo elemento que lo reemplace. Pero sabemos que el árbol resultante tendrá un mínimo más grande que el anterior o máximo más chico. Por lo que la postcondición necesaria para eliminar un elemento es la siguiente:

```
{-@ deletePostCondition:: (Ord a) => tf :  ABB a ->
  t : ABB a -> Bool @-}
deletePostCondition :: (Ord a) => Tree a -> Tree a -> Bool
deletePostCondition tf t
    | notEmpty tf && notEmpty t =
        (greatest tf == greatest t || greatest tf < greatest t)
        && (lowest tf == lowest t || lowest tf > lowest t)
    | otherwise = isEmpty tf
{-@ inline deletePostCondition @-}
```

donde el primer elemento es el árbol final y el segundo el inicial. Como podemos observar, la función permite que los árboles sean vacíos. Esto se debe a que el elemento puede no estar presente en el árbol al momento de eliminarlo. Para contemplar este caso y que la función `delete` termine, será necesario recorrer todo el árbol. La guarda `otherwise` le termina de dar completitud a la función porque si no se cumple que los árboles no sean vacíos, entonces requiere que el árbol final lo sea.

Al mismo tiempo, podemos observar que no se contemplan los casos en que uno de los árboles es vacío y el otro no. Esto se debe a que esta postcondición se utiliza para reconstruir el árbol una vez ejecutado el paso siguiente de la función delete. En ese sentido, solo nos importa saber el rango de valores del mínimo y máximo del árbol resultante. En caso de que el inicial sea vacío, el final será igual y, en el caso de que el final no tenga elementos, no tendrá ni mínimo ni máximo, por lo que el resultado final siempre es que el árbol es vacío.

Además de la postcondición, ciertos casos requerirán combinar las ramas restantes en caso de eliminar un nodo intermedio del árbol, por lo que será necesario implementar una función `merge`. El resultado de esta función deberá respetar la postcondición establecida por `delete` porque será llamada por la misma. A continuación podemos ver la postcondición de la función `merge` la cual nos asegura que la misma cumpla con lo anterior:

```
  {-@ mergePostCondition :: (Ord a) => r :  ABB a ->
  l : ABB a -> f : ABB a -> Bool @-}
  mergePostCondition :: (Ord a) => Tree a -> Tree a -> Tree a ->
   Bool
  mergePostCondition r l f
      | notEmpty l && notEmpty r && notEmpty f =
        lowest f >= (lowest l) && greatest f <= (greatest r)
      | notEmpty l && isEmpty r && notEmpty f =
        lowest f == lowest l && greatest f == greatest l
      | isEmpty l && notEmpty r && notEmpty f =
        lowest f == lowest r && greatest f == greatest r
      | otherwise = isEmpty f
  {-@ inline mergePostCondition @-}
```

En este caso, es posible indicar que el resultado de la función mantiene los elementos de máximo y mínimo, pero esto genera un problema en el caso de que las ramas no sean vacías. El compilador no es capaz de darse cuenta de que estos valores no cambian. Sin embargo, sí es capaz de detectar que están dentro del rango establecido por el mínimo de `l` y máximo de `r`, lo cual es suficiente para satisfacer la postcondición de la función `delete`.

Es así que la función `merge` se implementa de la siguiente manera:

```
  {-@ merge :: (Ord a) => r : ABB a  ->
    l : { ABB a | (notEmpty l && notEmpty r) =>
      greatest l < lowest r } ->
    f : { ABB a | mergePostCondition r l f } @-}
merge :: (Ord a) => Tree a -> Tree a -> Tree a
merge r@(Node v2 l2 r2) l@(Node _ _ _) = Node v2 (merge l2 l) r2
merge (Nil) l = l
merge r (Nil) = r
```

Por último replanteamos la función `delete` inicial haciendo uso de todas las anteriores:

```
{-@ delete :: t : ABB a -> v: a ->
  tf: { ABB a | deletePostCondition tf t } @-}
delete :: (Ord a) => Tree a -> a -> Tree a
delete (Nil) _ = Nil
delete t@(Node v l r) k
    | v > k = Node v (delete l k) r
    | v < k = Node v l (delete r k)
    | otherwise = merge r l
```

3.2. Implementación internalista

En una implementación internalista el árbol es ABB por construcción, esto quiere decir que, en el tipo del árbol, el constructor del nodo define el dominio de valores de sus ramas, las cuales admitirán valores menores o mayores al valor del nodo padre.

Como vimos en la sección de preliminares, para la implementación internalista LH nos permite restringir los GADT con los subdominios que nosotros queramos. Es así que plantearemos las siguientes modificaciones a la estructura de árbol inicial:

```
data ABB a where
  Node :: Ord a => a -> ABB a -> ABB a -> ABB a
  Nil :: Ord a => ABB a

{-@
data ABB a where
  Node :: Ord a => x: a -> ABB { vv : a |  x > vv } ->
    ABB { vv: a | x < vv } -> ABB a
  Nil :: Ord a => ABB a
@-}
```

En este caso, la primera definición de ABB corresponde al código Haskell del GADT, la segunda al código refinado, el cual restringe el dominio de las ramas. En la rama izquierda, los valores del subárbol serán menores al nodo raíz y en el de la derecha mayores. A diferencia de la implementación externalista no existe un subtipo de los árboles que sean ABB, es por eso que renombramos el tipo, ya que todos tendrán que cumplir las restricciones establecidas a las ramas.

3.2.1. Insert

En este caso, la condición del árbol será estructural, por lo que basta con recorrerlo hasta el lugar correspondiente y no será necesario agregar una función de postcondición. Esto se debe a que LH conoce el dominio de los subárboles, por ende su máximo o mínimo y es así que puede comparar el valor del nodo con el valor a insertar. La única información que necesitamos verificar para poder realizar esta operación de forma exitosa es que el valor sea mayor o menor o igual al elemento del nodo. A continuación podemos ver la función `insert`:

```
{-@ insert :: (Ord a) => ABB a -> a -> ABB a  @-}
insert :: (Ord a) => ABB a -> a -> ABB a
insert (Nil) v = Node v Nil Nil
insert t@(Node v2 l r) v
    | v < v2 = Node v2 (insert l v) r
    | v > v2 = Node v2 l (insert r v)
    | otherwise = t
```

Como podemos observar, la simplificación de la función es notable.

3.2.2. Delete

En este caso ya no será tan fácil remover un elemento del árbol, principalmente porque es necesario reordenar el árbol luego de borrado el elemento. Es así que en cuanto encontremos el nodo a eliminar, para hacerlo de forma similar a como lo hicimos en la implementación externalista llamaremos a la función `merge`:

```
delete :: (Ord a) => ABB a -> a -> ABB a
delete (Nil) v = Node v Nil Nil
delete (Node v2 l r) v
    | v < v2 = Node v2 (insert l v) r
    | v > v2 = Node v2 l (insert r v)
    | otherwise = merge v r l
```

donde la función `merge` se utiliza para combinar los dos ABB correspondientes a las ramas del árbol con elemento eliminado en uno:

```
{-@ merge :: (Ord a) =>  v : a -> ABB {vv: a | v < vv} ->
  ABB {vv: a | vv < v } -> ABB a @-}
merge :: (Ord a) => a -> ABB a -> ABB a -> ABB a
merge v (Nil) t = t
merge v (Node v2 l r) t = (Node v2 (merge v l t) r)
```

En este caso, fue necesario pasar un elemento más, el valor eliminado, ya que constituye el límite entre los valores mayores y menores. Sin este elemento, no podríamos indicarle a LH que una rama es menor que otra. Esto sí era posible en la implementación externalista porque teníamos el concepto de máximo y mínimo. Aun así, las reducciones de código de esta implementación son grandes, puesto que reduce la cantidad de líneas a la mitad.

3.3. Comparación entre las implementaciones internalista y externalista

Ambas implementaciones comparten la mayoría del código, por lo que las diferencias entre una y otra no radican tanto en el código Haskell, sino en las precondiciones y postcondición que hay que utilizar entre una y la otra.

Mientras la implementación externalista se apoya en la definición de máximo y mínimo para definir a los ABB, la implementación internalista lo hace estructuralmente, lo cual permite utilizar menor cantidad de pre y postcondiciones. Sin embargo, la implementación internalista por sí sola no permite manejar la noción de extremos sin utilizar el nodo padre a las ramas. Esto se podría arreglar fácilmente implementando las funciones máximo y mínimo.

Para finalizar, por lo mencionado anteriormente, la cantidad de líneas de código utilizado en la implementación internalista es la mitad que la externalista.

Capítulo 4

Implementación de AVL

En este capítulo se detalla la implementación realizada de los árboles de Adelson-Velskii y Landis (AVL), los cuales son árboles binarios de búsqueda y tienen la particularidad de ser balanceados. Esta condición establece que el valor absoluto de la diferencia de altura entre la rama derecha e izquierda de todo subárbol no puede ser mayor a uno.

Para esta implementación (de León, 2024b) se optó por un abordaje internalista y nos basamos en el tutorial propuesto en la página de LiquidHaskel (Ranjit Jhala, s.f.). En la Sección 4.5.2 profundizaremos en las similitudes y diferencias de las implementaciones.

4.1. Modificaciones estructurales

Si bien comentamos que los árboles AVL son ABB, hay una particularidad en la condición de balanceo, ya que crea la necesidad de tener acceso a la altura del árbol de forma continua. Si utilizáramos la función altura que definimos para los ABB, ejecutaríamos una operación $O(n)$ cada vez que la utilicemos. Tanto la función insert y delete necesitarán chequear la altura una vez realizada la operación, por lo que el costo de estas sería de $O(nlog(n))$. Este orden es mayor al previsto por los creadores de la estructura, por lo que realizaremos modificaciones a la estructura del árbol que nos permitan alcanzar un orden menor.

Presentadas estas condiciones, necesitamos definir una forma más eficiente de consultar la altura. Para eso agregaremos un atributo `h` al constructor `Node`, el cual corresponderá en valor a la altura del árbol. Además de agregar el atributo LH nos permite asignar el mismo valor que tendría la evaluación de la función altura y forzar que este coincida con ese valor en tiempo de compilación. La estructura final del árbol será de la siguiente manera:

```haskell
data AVL a where
  Node :: Ord a => a -> AVL a -> AVL a -> Int -> AVL a
  Nil  :: Ord a => AVL a

{-@
data AVL a where
  Node :: Ord a => x: a ->
    l: AVL { vv : a |  x > vv } ->
    r: AVL { vv : a | x < vv } ->
    h: { Int | h == 1 + (max (height l) (height r)) } ->
    f: AVL a
  Nil :: Ord a => AVL a
@-}
```

Como podemos ver, cambiamos el nombre ABB por AVL. Al mismo tiempo, vemos que el atributo `h` tiene la condición de ser igual a la altura en tiempo de compilación, por lo que todas las asignaciones del mismo tendrán que cumplir esa condición.

Una pregunta que nos puede surgir luego de ver esta implementación es: ¿por qué no usar que la altura del árbol `f` es igual a `h`, en vez de restringir el atributo? Esto se debe a la forma como LH evalúa los GADT; cuando agregamos condiciones al tipo resultante de un GADT LH asume que esa condición se cumple.

Es así que si nosotros planteamos la siguiente definición:

```haskell
{-@
data AVL a where
  Node :: Ord a => x: a ->
    l: AVL { vv : a |  x > vv } ->
    r: AVL { vv : a | x < vv } ->
    h: Int  ->
    f: { AVL a | height f == h }
  Nil :: Ord a => AVL a
@-}
```

entonces este ejemplo es correcto para LH:

```haskell
{-@ test :: AVL Int @-}
test :: AVL Int
test = Node 6 Nil Nil 99
```

Sin duda, esta afirmación es errónea. Por lo tanto, la única manera de restringir la altura de otro modo es replicar el contenido de la función `height` y añadirlo como una condición. Finalmente, creamos una función llamada `getHeight` que facilita el acceso a la altura de un árbol en un tiempo constante de O(1).

```haskell
{-@ getHeight :: t : AVL a -> {h: Int | height t == h }@-}
getHeight :: AVL a -> Int
getHeight Nil = 0
getHeight (Node _ _ _ h) = h
{-@ measure getHeight @-}
```

Esta función tiene como postcondición que su resultado equivale a la altura del árbol. Durante este capítulo utilizaremos de distintas formas `height` y `getHeight`, lo importante es siempre utilizar getHeight cuando el código que estamos escribiendo se ejecute. Muchas veces definimos funciones de pre o postcondiciones que nunca son ejecutadas por el código, sino que LH las utiliza para chequear y poder corroborar que el código sea correcto. En esos casos podremos utilizar `height` sin comprometer la eficiencia al momento de ejecutar la función.

4.2. Peso

Como mencionamos antes, al construir esta estructura seguimos un enfoque similar al utilizado en los ABB, aunque añadimos funciones adicionales relacionadas con la diferencia entre la rama izquierda y derecha del árbol, que llamaremos "peso" o `weight` por su nombre en inglés. Para determinar la inclinación del árbol, solo necesitamos conocer el peso de dos árboles independientes. Esto lo lograremos mediante la siguiente función:

```haskell
{-@ inline weightLR @-}
weightLR :: (Ord a) => AVL a -> AVL a-> Int
weightLR l r = getHeight l - getHeight r
```

Como podemos ver, nuestra nueva estructura nos permite calcular la diferencia de alturas en tiempo O(1). Teniendo una forma de obtener el peso entre dos árboles independientes, podremos definir la condición de balance de un árbol de la siguiente manera:

```haskell
{-@ inline balanced @-}
balanced :: (Ord a) => AVL a -> AVL a -> Bool
balanced t1 t2 = abs (weightLR t1 t2) <= 1
```

Volvamos a nuestra estructura, en este caso estamos buscando condicionar las ramas de nuestra estructura de forma tal que el constructor admita únicamente dos subárboles que son compatibles. Esto podemos conseguirlo limitando una rama, en nuestro caso la derecha, de forma que la misma tenga que cumplir la condición de balanceo con la rama izquierda. Una vez conseguido esto, tenemos las funciones necesarias para satisfacer la definición de AVL en nuestra estructura. A continuación podemos ver el resultado final del código en LH:

```
{-@
data AVL a where
  Node :: Ord a => x: a ->
    l: AVL { vv : a |  x > vv } ->
    r: { AVL { vv: a | x < vv }| balanced l r } ->
    h: { Int | h == 1 + (max (height l) (height r)) } ->
    AVL a
  Nil :: Ord a => AVL a
@-}
```

Al agregar la condición de balanceo, tenemos el tipo AVL que cumple con todos los requerimientos de la estructura.

Por último, en el futuro nos será de ayuda calcular el peso de un AVL en particular. La necesidad detrás de esto es poder chequear el balanceo de un árbol sin tener que acceder a sus ramas para hacerlo. Para eso definimos la función `weight` implementada a continuación:

```
{-@ measure weight @-}
{-@ weight :: AVL a -> w : { Int | w >= -1 && w <= 1}@-}
weight :: (Ord a) => AVL a -> Int
weight (Nil) = 0
weight (Node _ l r _) = weightLR l r
```

Como podemos observar, el resultado de la función está acotado entre 1 y menos 1. Una pregunta que podría surgir es como podemos limitar la salida de la función entre -1 y 1. Dado que estamos usando la estructura AVL a la cual ya hemos acotado la diferencia de altura que pueden tener los subárboles de un AVL por ende esta condición se cumple y puede ser satisfecha por LH.

4.2.1. Insert

Al momento de realizar la inserción en un AVL y a diferencia de los ABB ya no podremos únicamente recorrer el árbol hasta encontrar el lugar donde insertar el elemento. Esto se debe a que cuando insertamos podemos desbalancear el árbol que anteriormente cumplía esta condición. Es por eso que debemos realizar algunas modificaciones a la función `insert` definida para los ABB y contemplar estos casos. Para terminar de insertar de forma correcta, será necesario volver a balancear el árbol de forma que vuelva a cumplir la condición. Es así que definimos nuestra función `insert` de la siguiente forma:

```haskell
 {-@ inline insertPC @-}
insertPC :: (Ord a) => AVL a -> AVL a -> Bool
insertPC f t = fh == th || fh == th + 1
  where
    th = height t
    fh = height f

{-@ insert :: (Ord a) -> t: AVL a -> a ->
  tf : { AVL a | insertPC tf t }
@-}
insert :: (Ord a) => AVL a -> a -> AVL a
insert Nil v = Node v Nil Nil 1
insert t@(Node v l r _) v2
  | v > v2 = balance v (insert l v2) r
  | v < v2 = balance v l (insert r v2)
  | otherwise = t
```

Como se puede apreciar, incluimos la función `balance` la cual rebalancea el árbol en caso de que ya no cumpliera la condición. Además de esto, fue determinante agregar una postcondición para la inserción, ya que para mantener el equilibrio del árbol, necesitamos información sobre su altura para unificarlo con los elementos restantes posteriormente.

4.3. Balanceo

En esta sección profundizaremos en la implementación de la función de balanceo. Para llevar a cabo esta operación, emplearemos una función que combine dos árboles y un elemento. No es necesario que estos estén desbalanceados, pero es importante para nosotros que la diferencia entre la altura de uno y otro subárbol no supere 2. Esta cifra se justifica por la diferencia máxima permitida entre dos ramas de un AVL cuando se inserta un elemento adicional. Así, definimos la función `balance` de la siguiente manera:

```haskell
{-@
balance :: x: a
  -> l : AVL { v: a | v < x }
  -> r: { AVL { v: a | v > x } | abs (weightLR l r)  <= 2 }
  -> f : { AVL a | balancePC f l r }
@-}
balance :: (Ord a) => a -> AVL a -> AVL a -> AVL a
balance v l r
  | td == 2 && wl == 0 = balL0 v l r
  | td == 2 && wl == 1 = balLL v l r
  | td == 2 && wl == -1 = balLR v l r
  | td == -2 && wr == 0 = balR0 v l r
  | td == -2 && wr == 1 = balRL v l r
  | td == -2 && wr == -1 = balRR v l r
  | otherwise = Node v l r (1 + max (getHeight l) (getHeight r))
  where
    td = weightLR l r
    wl = weight l
    wr = weight r
```

Como podemos ver, las limitaciones que hicimos sobre el dominio de la función son de gran ayuda, ya que como consecuencia de ello reducimos la cantidad de casos a un número conciso donde dividimos los casos para ejecutar una rotación distinta dependiendo del estado del árbol. Los casos donde se ejecutan rotaciones son aquellos en que la diferencia de las alturas no cumple la condición de balanceo. Si nos centramos en esos casos, podremos observar 6 rotaciones distintas que dependen de la inclinación del árbol y sus ramas.

Otro punto a destacar es la utilización de una postcondición. Como sabemos que la altura del árbol va a cambiar al momento de realizar el balanceo, implementamos la función `balancePC` de la siguiente forma:

```haskell
{-@ inline balancePC @-}
balancePC :: (Ord a) => AVL a -> AVL a -> AVL a -> Bool
balancePC f l r = hf == 2 + minh || hf == 1 + maxh
  where
    maxh = maxHeight l r
    minh = minHeight l r
    hf = height f
```

Esta condición nos permite unificar el árbol balanceado con la altura. Esto se debe a que en el mejor de los casos los elementos son compatibles, por lo que el resultado del balanceo es simplemente aplicar el constructor `Node` sin hacer ninguna rotación. En el caso en que alguna rotación es necesaria, entonces el árbol resultante será de la altura mínima entre las ramas más 2. Esto se debe a que la diferencia entre las alturas es 2. El procedimiento de balanceo cambia la diferencia de altura y la altura mínima; ahora se incrementa en 1, y al

agregar el elemento del nodo padre, hace que sumen 2. Podremos ver en particular cada ejemplo en la siguiente sección donde describiremos las rotaciones implementadas.

4.4. Rotaciones

Como la construcción de los árboles será una tarea que repetiremos muchas veces, tendremos la necesidad de calcular el atributo altura cada vez que lo utilicemos. Para no tener que estar escribiendo el cuerpo de la función `height` una y otra vez utilizaremos una función auxiliar `makeT`. La implementación la podemos ver a continuación:

```
{-@ makeT ::  v : a ->
  l : AVL { v2 : a | v2 < v} ->
  r: { AVL { v2 : a | v2 > v} | balanced l r } ->
  f : { AVL a | height f == (1 + maxHeight l r )}
@-}
makeT :: (Ord a) => a -> AVL a -> AVL a -> AVL a
makeT  v l r  = Node v l r (1 + max (getHeight l) (getHeight r))
```

Una vez introducida esta función auxiliar, procedemos a analizar individualmente cada caso de las rotaciones a realizar. Partiendo desde el nodo padre, la elección de la rotación dependerá de la inclinación del subárbol con mayor peso. En cada caso, hay un denominador común: dado que la diferencia de altura es 2, es necesario realizar una rotación en el árbol de manera que se añadan elementos al subárbol de menor peso, restableciendo así la condición de balanceo.

En este proceso, utilizaremos siempre el ex-nodo padre para redefinir el subárbol, desplazando alguna subrama del subárbol con mayor peso. Entre las posibles rotaciones, identificamos dos casos principales: las rotaciones que requieren un único movimiento, conocidas como rotaciones simples, y aquellas que implican la combinación de dos movimientos para restaurar la condición de balanceo, denominadas rotaciones dobles.

4.4.1. Rotaciones simples

Separaremos los casos de las rotaciones simples en dos escenarios: aquellos en los que el peso recae sobre el árbol izquierdo y su contraparte, donde el peso se encuentra en el árbol derecho. Ambos comparten la similitud de que, en la rama más pesada, el árbol se encuentra en equilibrio, lo que implica que la diferencia de altura entre las subramas es igual a cero.

Peso sobre la rama izquierda

En el caso en que el subárbol izquierdo tenga más peso que el derecho, tendremos que realizar la siguiente rotación:

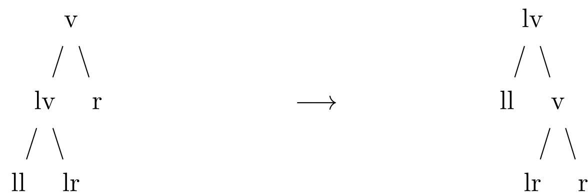


Figura 4.1: Rotación del árbol de izquierda a derecha en equilibrio

Los subárboles ll y lr tendrán alturas idénticas, por lo que la estrategia será desplazar el subárbol derecho lr y fusionarlo con el árbol derecho r mediante el nodo padre v. Por lo tanto, la operación en código será de la siguiente forma:

```
{-@ balL0 :: x:a
          -> l:{ AVL {v:a | v < x} | weight l == 0 }
          -> r:{ AVL {v:a | v > x} | weightLR l r == 2 }
          -> { t: AVL a | height t == height l + 1 }
  @-}
balL0 v (Node lv ll lr _) r = makeT lv ll (makeT v lr r)
```

El resultado obtenido será un árbol que tenga altura l + 1, sin embargo, esta altura estará del lado derecho.

Peso sobre la rama derecha

Este caso es muy similar al anterior; la única diferencia será la dirección en la que se hace la rotación.

```
{-@ balR0 :: x:a
          -> l: AVL {v:a | v < x}
          -> r:{ AVL {v:a | v > x} | weightLR l r == -2 &&
          ↪  weight r == 0 }
          -> { t: AVL a | height t == height r + 1 }
  @-}
balR0 :: (Ord a) => a -> AVL a -> AVL a -> AVL a
balR0 v l (Node rv rl rr _) = makeT rv (makeT v l rl) rr
```

4.4.2. Rotaciones dobles

En las rotaciones dobles tendremos la particularidad de que el subárbol que posee más peso no está en equilibrio, por lo que una única rotación ya no será suficiente para hacer cumplir la condición de los AVL. Por esto, dependiendo de la inclinación del subárbol, tendremos que realizar otra rotación más a la derecha o izquierda.

Subárbol izquierdo balanceado a la derecha

En el caso de que el peso del subárbol izquierdo esté balanceado a la derecha, entonces la rotación a realizar será la siguiente:

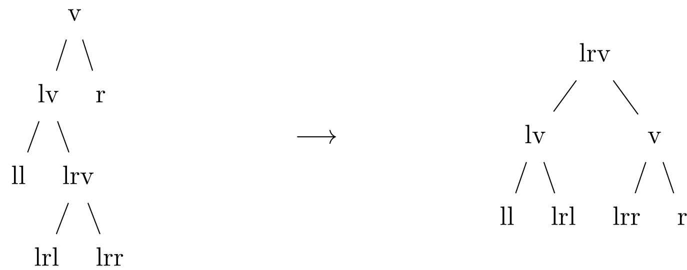


Figura 4.2: Rotación del árbol de izquierda a derecha con balanceo -1

En este caso, la estrategia implica desplazar el nodo del subárbol derecho lrv hacia una posición superior, convirtiéndolo en el nuevo nodo padre. Al hacer esta operación, debemos integrar los subárboles lrl y lrr a uno de los árboles restantes. El subárbol izquierdo se reconecta al árbol izquierdo a través de LV, mientras que el subárbol derecho se une al árbol derecho r mediante el exnodo padre v. De esa forma, la operación a realizar es la siguiente:

```
{-@ balLR :: x:a
          -> l:{ AVL { v:a | v < x } | weight l == -1 }
          -> r:{ AVL { v:a | v > x } | weightLR l r == 2 }
          -> { t : AVL a | height t == height l }
  @-}
balLR v (Node lv ll (Node lrv lrl lrr _) _) r
  = makeT lrv (makeT lv ll lrl) (makeT v lrr r)
```

Como podemos ver, en este caso utilizar `measures` en estas estructuras es de gran utilidad. Nos permite utilizar pattern matching en profundidad, cosa que no sería posible sin LH.

Subárbol izquierdo balanceado a la izquierda

En este caso la rotación a realizar será la siguiente:

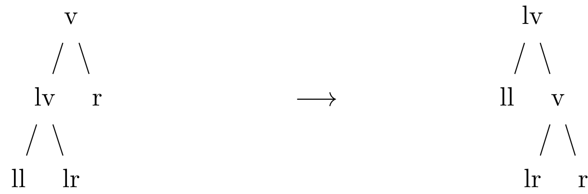


Figura 4.3: Rotación del árbol de izquierda a derecha con balanceo 1

Para esta rotación, el árbol izquierdo está balanceado hacia la izquierda. El procedimiento para este caso será igual al realizado cuando el árbol izquierdo está en equilibrio, con diferencia de que la altura resultante será menor a la de la operación anterior.

```
{-@ balLL :: x:a
          -> l:{ AVL { v:a | v < x } | weight l == 1 }
          -> r:{ AVL { v:a | v > x } | weightLR l r == 2 }
          -> { t : AVL a | height t == height l }
  @-}
balLL :: (Ord a) => a -> AVL a -> AVL a -> AVL a
balLL v (Node lv ll lr _) r = makeT lv ll (makeT v lr r)
```

Rotaciones de derecha a izquierda

En los casos en que el desbalanceo esté inclinado al lado derecho, las operaciones serán análogas a las descritas anteriormente. Con la diferencia de que la orientación de los movimientos cambia, en vez de mover de izquierda a derecha se realiza de derecha a izquierda. A continuación podemos ver el código de ambas rotaciones:

```
{-@ balRR :: x:a
        -> l: AVL { v: a | v < x }
        -> r:{AVL { v: a | v > x } | weightLR l r == -2 &&
        ↪  weight r == -1 }
        -> {t: AVL a | height t == height r }
@-}
balRR :: (Ord a) => a -> AVL a -> AVL a -> AVL a
balRR v l (Node rv rl rr _) = makeT rv (makeT v l rl) rr

{-@ balRL :: x:a
        -> l: AVL { v: a | v < x }
        -> r:{AVL { v: a | v > x } | weightLR l r == -2 &&
        ↪  weight r == 1 }
        -> { t: AVL a | height t == height r }
@-}
balRL :: (Ord a) => a -> AVL a -> AVL a -> AVL a
balRL v l (Node rv (Node rlv rll rlr _) rr _) =
  makeT rlv (makeT v l rll) (makeT rv rlr rr)
```

Como podemos ver, ambas son muy similares a las definiciones que mostramos en la sección anterior.

4.5. Delete

La función `delete` en un AVL tiene la misma particularidad de la función `insert` y no solo será necesario mantener el orden del árbol sino su balanceo. Es por eso que la función `delete` utilizada para los ABB ya no nos sirve y es necesario agregar elementos nuevos.

Comenzaremos definiendo la postcondición de la función `delete`. En este caso, eliminar un elemento de la estructura significará que la altura de la misma se mantendrá igual o decrecerá en uno. Por lo tanto, `delete` deberá cumplir la siguiente postcondición:

```
{-@ inline deletePC @-}
deletePC :: (Ord a) => AVL a -> AVL a -> Bool
deletePC f t =  height f == height t || height f == height t - 1
```

Una vez que se ha establecido esta postcondición, ahora contamos con información sobre el resultado de la función, lo cual nos permite tomar decisiones basadas en si el árbol ha perdido su balance o no, y ejecutar la rotación correspondiente en consecuencia. Para lograrlo, vamos a introducir algunas adiciones a la función `delete` previamente definida para los ABB, incorporando la función `balance`. Además, tendremos que redefinir `merge` de manera que conserve el orden y asegure el balance de la manera más eficiente posible.

A continuación, detallamos las modificaciones que se llevarán a cabo en la función `delete`

```
{-@ delete :: (Ord a) => t : AVL a -> v : a ->
  f: { AVL a | deletePC f t }
@-}
delete :: (Ord a) => AVL a -> a -> AVL a
delete Nil _ = Nil
delete t@(Node v l r _) v2
  | v > v2 = balance v (delete l v2) r
  | v < v2 = balance v l (delete r v2)
  | otherwise = merge v l r
```

Como podemos ver, esta función no tiene una gran complejidad, aunque requerirá redefinir ciertos componentes de las funciones `merge` y `balance` que veremos a continuación.

4.5.1. Merge

Comenzaremos implementando los cambios a la función `merge`. En estructuras anteriores, simplemente nos limitábamos a recorrer la rama derecha hasta llegar al nodo con menor valor y agregarle la rama izquierda como hijo. En este caso esto ya no es posible debido a que necesitamos mantener la condición de balanceo. Por esto necesitaremos buscar el mayor valor de la rama izquierda y utilizarlo como nodo padre para unir ambas ramas.

Para obtener el par dado por el mayor valor de un árbol junto con el mismo árbol sin este elemento, utilizaremos la función `maxValue` que recorre un AVL en busca de su mayor elemento y devuelve un par elemento árbol.

```
{-@ maxValue :: (Ord a) =>  t : { AVL a | notEmpty t } ->
  p :{ (v::a, AVL { vv : a | vv < v }) | deletePC (snd p) t }
@-}
maxValue ::(Ord a) => AVL a -> (a, AVL a)
maxValue (Node v l Nil _) = (v, l)
maxValue (Node v l r h) = (v3, balance v l r2)
  where
    (v3, r2) = maxValue r
```

Como podemos ver, el árbol no puede ser vacío porque en ese caso no tendría sentido realizar la operación. Para mantener la restricción de orden podemos utilizar los `::` dentro del par para darle nombre al elemento v. Sin embargo, no es posible definir un nombre para el segundo componente y utilizarlo en la condición de `deletePC`, por lo tanto, utilizamos la función `snd` de los pares que nos provee el preludio.

Una vez implementada esta función podremos escribir la función `merge`:

```
{-@ merge :: (Ord a) =>  v : a -> l: AVL {vv: a | v > vv } ->
  r : { AVL {vv: a | vv > v} | balanced l r } ->
  f: { AVL a | mergePC l r f }
@-}
merge :: (Ord a) => a -> AVL a -> AVL a -> AVL a
merge _ (Nil) t = t
merge _ l r = balance mv mt r
  where
    (mv, mt) = maxValue l
```

Sin embargo, esta función no compila para LH. Esto se debe a la incompatibilidad que hay entre las postcondiciones definidas para la función *merge* y *balance*. Para entender un poco mejor esto veamos ambas postcondiciones:

```
{-@ inline mergePC @-}
{-@  mergePC :: (Ord a) => l: AVL a ->
  r : { AVL a | balanced l r } ->
  f: AVL a -> Bool
@-}
mergePC :: (Ord a) => AVL a -> AVL a -> AVL a -> Bool
mergePC l r f
  | td == 1 = hl == fh || hl + 1 == fh
  | otherwise = rh == fh || rh + 1 == fh
  where
    td = weightLR l r
    rh = height r
    hl = height l
    fh = height f
```

Lo primero que podemos observar es que el árbol utilizado para definir la altura final depende de la diferencia que haya entre las ramas. Esto es una diferencia bastante grande respecto a las postcondiciones anteriores, porque solíamos definir las mismas en función de *maxHeight*. Si la altura de la rama izquierda es mayor, entonces es fácil utilizar esta propiedad para definir la altura final. Sin embargo, si esta no lo es, al descomponer el árbol izquierdo, la altura final ya no depende únicamente de *maxHeight*. Esto se debe a que en el caso antes mencionado se podría dar un desbalanceo que provoque la ejecución de rotaciones. Por ende la altura final ya no depende de que una sea mayor a la otra, sino de a dónde esté inclinado el árbol inicial.

Como podemos ver, esta función tiene una mayor precisión en sus condiciones que las anteriormente implementadas. Al requerir un mayor grado de precisión, la postcondición de `balance` se torna muy general, ya que hay ciertos casos que son posibles para `balancePC` que no lo son para `mergePC`. Esto implica que LH no pueda unificar las definiciones tal y como están por el hecho de que hay casos que cumplen la postcondición de `balance` pero no la de `merge`. Para lograr

que LH pueda deducir correctamente el resultado del balanceo, necesitaremos agregar una casuística similar a la de *mergePC*. A continuación, podemos ver la implementación de la nueva postcondición de balance.

```
{-@ inline balancePC @-}
{-@
balancePC :: l : AVL a ->
 r:{ AVL a| weightLR l r  <= 2 && weightLR l r  >= -2} ->
 f : AVL a -> Bool
@-}
balancePC :: (Ord a) => AVL a -> AVL a -> AVL a -> Bool
balancePC f l r
  | td > 0 && wl == 0 = height f == hl + 1
  | td > 0 = height f == hr + 2
  | td < 0 && wr == 0 = height f == hr + 1
  | td < 0 = height f == hl + 2
  | otherwise = height f == 1 + maxHeight l r
  where
    td = hl - hr
    hl = height l
    hr = height r
    wl = weight l
    wr = weight r
```

Este cambio en las condiciones de la función, nos brinda más precisión sobre la altura resultante del árbol luego del balanceo. Gracias a esto, la función `merge` como la definimos será aceptada por LH. Sin embargo, esto requirió aumentar la cantidad de líneas de código y la complejidad de nuestra solución anterior. Además, que el error obtenido cuando la función `balance` no podía unificar las condiciones no es nada claro, ya que simplemente dice que no se puede cumplir con las postcondiciones de la función `mergePC`. Fue necesario probar caso por caso para visualizar el problema de fondo y poder obtener una solución.

4.5.2. Comparación entre implementaciones

Si comparamos la solución presentada en nuestro trabajo con la que provee el tutorial de LiquidHaskell la diferencia principal se encuentra en la cantidad de funciones auxiliares que usamos. En este trabajo, optamos por la utilización de menos funciones auxiliares para poder ver en particular cómo se descompone la altura del árbol y las desigualdades de sus ramas en cada caso.

Mientras que en la implementación presentada por el tutorial se suele acudir a funciones auxiliares que en muchos casos no explicitaban directamente el problema que estábamos tratando de resolver. Es por esto que optamos por utilizar la altura o diferencia de la misma como punto de comparación.

Capítulo 5

Implementación de BraunTrees

Los BraunTrees (Hoogerwoord, 1992) son una estructura arborescente que nos permite representar arreglos funcionales. Estos tienen la particularidad de aumentar o disminuir en tamaño con un bajo costo computacional, además de implementar de forma muy eficiente la búsqueda de índices dentro del árbol.

Esta estructura implementa un método de búsqueda diferente a los anteriores. En lugar de ordenar los elementos según su relación con el nodo padre, se emplea un índice específico. Dicho índice señala la posición dentro del árbol que se desea buscar. Para navegar por el árbol, utilizamos la representación binaria del índice, lo que nos permite tomar decisiones sobre qué rama seguir o determinar si hemos alcanzado el final de la búsqueda.

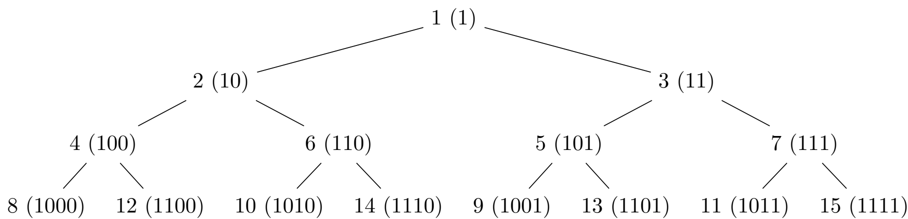


Figura 5.1: Ejemplo de un arbol BraunTree con sus respectivos índices, entre paréntesis su representación binaria

Por lo tanto, si queremos buscar el elemento 13 en el árbol de la Figura 5.1, el procedimiento de búsqueda implica analizar su representación binaria de derecha a izquierda. Cada bit de esta representación indica descender un nivel dentro del árbol: un bit con valor 0 nos lleva por la rama izquierda, mientras que un bit con valor 1 nos guía por la rama derecha. Al alcanzar el bit con valor 1 de mayor precedencia, identificamos la posición exacta dentro del árbol que

estábamos buscando. Como podemos ver en el ejemplo, todos los elementos pares se encuentran del lado izquierdo y todos los elementos impares se encuentran del lado derecho; esto es, porque poseen el dígito de menor precedencia en 1. La noción de paridad será de gran importancia y, como ya podemos ver, el árbol se divide en ramas con elementos pares a un lado e impares al otro.

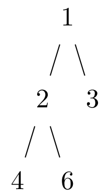
Dadas las características de la estructura y que existen implementaciones en otros lenguajes con propiedades similares a LH (Nipkow y Sewell, 2020) nos pareció de interés realizar nuestra propia implementación (de León, 2024c) de BraunTrees a modo de comparar la herramienta con las demás.

5.1. Estructura

Una vez definido el concepto de BraunTrees nos interesa plantear la estructura base que utilizamos. En este caso es similar a la utilizada en capítulos anteriores para representar árboles:

```
data Tree a where
  Node :: (Eq a) => a -> Tree a -> Tree a -> Tree a
  Nil :: Tree a
```

Con este punto de partida en mente, será necesario agregar propiedades de balanceo. Sin embargo, a diferencia de los AVLs no utilizaremos la altura como principal atributo para calcularla. Esto se debe a un requerimiento estructural de los BraunTrees. Si bien la condición utilizada en los AVL se cumple en los BraunTrees la misma no tiene sentido en este caso. Si nosotros quisiéramos representar el balanceo con la altura nos encontraríamos con el siguiente caso que es válido:



Árbol balanceado por la altura

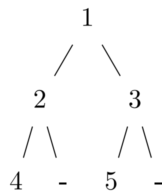
Este árbol se considera balanceado; sin embargo, el mismo no cumple con la definición de BraunTree. El usuario podrá realizar la operación `lookup t 6`, sin embargo, no podrá realizar `lookup t 5`. El balanceo en los BraunTrees no solo implica que la diferencia de alturas entre las subramas sea menor o igual a 1, sino que además debe restringir la cantidad de elementos que tienen las mismas. Es así que definimos `nodeCount`, la función que cuenta la cantidad de nodos de un árbol de la siguiente manera:

```haskell
{-@ nodeCount :: t : Tree a -> n : Nat @-}
nodeCount :: Tree a -> Int
nodeCount (Node _ l r) = 1 + nodeCount l + nodeCount r
nodeCount Nil = 0
{-@ measure nodeCount @-}
```

Una vez definida la función `nodeCount` podemos escribir la condición de balanceado y verificación estructural para que un árbol sea un BraunTree:

```haskell
{-@ braun :: Tree a -> Bool @-}
braun :: Tree a -> Bool
braun (Node _ l r) = (nodeCount l == nodeCount r ||
    nodeCount l == nodeCount r + 1) && (braun l) && (braun r)
braun Nil = True
{-@ reflect braun @-}
```

Podemos observar que la cantidad de nodos en ambas ramas será igual, o la rama izquierda tendrá un nodo más. Esta propiedad limita a la rama derecha, impidiendo que contenga más elementos que la izquierda. Esto se debe a que, al agregar un nodo a la derecha, estamos añadiendo un índice cuya representación binaria tiene un 1 en su bit menos significativo. El índice con la misma representación binaria pero con un 0 en su bit menos significativo es menor, lo que implica que debe existir en el árbol para evitar la inconsistencia que queremos corregir. Así se explica por qué la rama izquierda tiene prioridad sobre la derecha, lo que se refleja en la condición de balanceo. Por lo tanto, el árbol representado en la imagen anterior tendrá la siguiente estructura:



Árbol balanceado por la cantidad de nodos

Al definir la condición de balanceo de esta manera, nos aseguramos que haya un elemento del árbol para cada posición del arreglo y evitamos que el caso anterior inválido no sea posible de construir.

5.1.1. Subtipos

Una vez definida la condición `braun` resta definir una estructura que la cumpla; para esto utilizaremos la directiva type. Además de esto, nos interesa agregar

algunos subtipos que nos serán de utilidad más adelante. Iniciaremos definiendo el tipo del arreglo o `Array` a continuación:

```
type Array a = Tree a

{-@
type Array a = { t : Tree a | braun t }
@-}
```

Nos interesa limitar la cantidad de elementos en un arreglo, por lo que construiremos el tipo de los arreglos de largo definido o `ArrayN`

```
{-@ type ArrayN a N = { arr : Array a | nodeCount arr == N } @-}
```

En ciertos casos nos interesa saber que la cantidad de nodos de un arreglo es igual o uno más al largo que le pasamos y para eso utilizamos `ArrayL`

```
{-@ type ArrayL a N = {
  arr: Array a | nodeCount arr == N || nodeCount arr == N + 1
} @-}
```

Además del caso anterior, nos interesa saber cuándo un arreglo tiene cantidad de nodos menor o igual a `n` y su análogo para la operación mayor. Agregamos entonces los subtipos `ArrayLE` y `ArrayGE` que representan los arreglos que cumplen esas condiciones.

```
{-@
type ArrayLE a N = { arr : Array a | nodeCount arr <= N }
type ArrayGE a N = { arr : Array a | nodeCount arr >= N }
@-}
```

Esta implementación de BraunTrees por definición es externalista; a diferencia de las estructuras anteriores (ABB y AVL), los BraunTrees no poseen una restricción sobre el conjunto de valores que tiene sus nodos. Lo que sí hay, es una restricción sobre la cantidad de elementos que tienen las ramas del árbol. Si quisiéramos realizar una implementación internalista del árbol, sería posible. Sin embargo, no habría diferencias tan grandes como en los casos anteriores. Dicha implementación quedó por fuera del alcance de este trabajo por lo que se plantea como una línea a seguir como trabajo a futuro.

Con los BraunTrees ya definidos, a continuación podemos ver la implementación. A modo de ejemplo usaremos los casos que vimos en la sección anterior:

```
{-@ fail test_not_work :: Array Int @-}
test_not_work = Node 1 l r
  where
    l = (Node 2 (Node 4 Nil Nil) (Node 6 Nil Nil))
    r = (Node 3 Nil Nil)

{-@ test_work :: Array Int @-}
test_work = Node 1 l r
  where
    l = (Node 2 (Node 4 Nil Nil) Nil)
    r = (Node 3 (Node 5 Nil Nil) Nil)
```

En el caso de `test_not_work` nos queremos asegurar que el árbol construido en la Figura 5.1 no es posible de representar. Debido a esto, tiene la instrucción `fail` que nos interesa que falle únicamente cuando el árbol sea posible de construir, ya que en ese caso habrá un error en la implementación. Para el caso de `text_work` nos interesa chequear que podamos construir un árbol de 5 nodos de forma correcta.

Crear árboles a través de sus constructores puede llegar a ser tedioso, por lo que para facilitar la creación de los mismos se implementaron las siguientes funciones (*smart constructors*):

```
{-@ singleton :: a -> ArrayN a 1 @-}
{-@ reflect singleton @-}
singleton :: Eq a => a -> Array a
singleton x = Node x Nil Nil

{-@ makeT :: v: a -> l : Array a ->
  r : { Array a |
    nodeCount l == nodeCount r || nodeCount l == nodeCount r + 1
  } ->
  t : ArrayN a { 1 + nodeCount l + nodeCount r }
@-}
{-@ reflect makeT @-}
makeT :: Eq a => a -> Array a -> Array a -> Array a
makeT v l r = Node v l r
```

5.2. Operaciones Básicas

Una vez que la estructura está definida y verificada para asegurar la correcta construcción de un BraunTree, podemos proceder a implementar sus operaciones básicas. Estas incluyen la búsqueda (lookup), actualización (update), largo (length) y la conversión de un arreglo a una lista (list).

Es así que los Arrays implementados a través de BraunTrees tendrán la siguiente interfaz:

```
lookup :: Int  Array a -> -> a
update :: Int -> a -> Array a -> Array a
length :: Array a -> Int
list  :: Array a -> [a]
```

5.2.1. Noción de paridad

Antes de poder implementar las operaciones mencionadas, es importante agregar a la lógica de predicados la noción de paridad. Esta tiene una gran importancia para los BraunTrees porque si un número es par o impar, significa que su bit menos significativo vale 0 o 1. Por ende, construimos la función par de la siguiente manera:

```
{-@ even :: n : Int -> Bool @-}
even :: Int -> Bool
even i = i `mod` 2 == 0
{-@ inline even @-}
```

Como podemos ver, en este caso no fue necesario utilizar ni `measure` ni `reflect`. Sin embargo, es de suma importancia que esta función pertenezca a la lógica de refinado, sin ella sería imposible recorrer los árboles.

5.2.2. Búsqueda

Para buscar un elemento en el árbol es necesario saber el índice al cual queremos acceder. Con el índice debemos tomar la decisión de avanzar en una dirección o retornar el elemento actual. Por definición, esta decisión dependerá de si el índice es par, impar o 1. Ya que definimos los arreglos con cantidad de nodos mayor a `n`, `ArrayGE a n` podemos escribir la función `lookup` de forma que siempre tengamos un resultado:

```
{-@ lookup1 :: { n: Nat | n > 0 } -> ArrayGE a n -> a @-}
lookup1 :: Int -> Array a -> a
lookup1 1 (Node v _ _) = v
lookup1 n (Node v l r)
  | even n = lookup1 (div n 2) l
  | otherwise = lookup1 (div n 2) r
{-@ reflect lookup1 @-}
```

La función se divide en dos casos distintos. El primer caso representa la condición de terminación de la función. Esta simplemente verifica si hemos alcanzado el final de la búsqueda mediante el chequeo de que `n` es igual a 1. Es

importante destacar que la función tiene como precondición que el índice sea mayor que 0, ya que se requiere al menos un bit en 1 para realizar una búsqueda válida en el árbol.

El otro caso, verifica si `n` es mayor que 1. Si lo es, procede a navegar a través del árbol: esto se logra al evaluar la paridad del índice `n`. Si el índice es par, avanzamos hacia la izquierda; de lo contrario, nos movemos hacia la derecha. Al considerar el bit menos significativo como 0 o 1, la división entera entre 2 simula la operación de desplazamiento a la derecha de un bit. Por ende, cada avance en el árbol implica una nueva llamada a la función dividiendo el índice entre 2. Aquí es donde entra la importancia de tener la función `even` en la lógica de predicados, sin ella LH nunca podría determinar que la división de n entre 2 es un número menor al `nodeCount` de los subárboles, lo cual es precondición de la función `lookup1`.

Si bien la estructura que estamos implementando en el fondo es un árbol, en su fachada es un arreglo, por lo que se desea emplear una firma de función más convencional. Es decir la función `lookup` usa `lookup1` con un índice aumentado en 1 ya que de esta manera el rango de indices disponibles en el arreglo sea de 0 a (n -1). A continuación podemos ver su implementación:

```
{-@ lookup :: n: Nat -> ArrayGE a { n + 1 } -> a @-}
lookup :: Int -> Array a -> a
lookup n arr = lookup1 (n + 1) arr
```

5.2.3. Actualizar

Si queremos actualizar un elemento de un BraunTree será necesario tener en cuenta dos casos. El primero es el de remplazar un valor en una posición que ya existe. El segundo, el de agregar un elemento al final del arreglo. Es así que construimos la función `update1` con una casuística similar a la función `lookup1`. Por un lado, recorreremos el árbol preguntando por la paridad del índice a actualizar. Si el índice es 1 entonces remplazaremos el elemento en la posición actual del árbol. Por otro lado permitiremos que la función extienda el arreglo; esto se realizará al encontrarnos con el constructor `Nil` en el recorrido del árbol.

Además de lo mencionado anteriormente, manejaremos pre y postcondiciones en la función. Para que no suceda el caso descrito en la Sección 4.1, será necesario limitar el índice para que no sea mayor a la cantidad de elementos del arreglo más 1 o viceversa. Si no restringimos este comportamiento, podríamos actualizar de forma tal que el arreglo quede inconsistente o intentemos acceder a un nivel el cual no forma parte del árbol. Es así que la precondición `ArrayGE a` { `n - 1` } limita a los arreglos a tener al menos la misma cantidad de elementos del índice menos 1.

Por otro lado, introduciremos el concepto de `ArrayNON1 a N I`, esta estructura no es opcional. Al construir un árbol en cada llamada recursiva, es necesario que el resultado de la función `update` tenga información de la cantidad de nodos

del árbol resultante. Por último, nos permitirá agregar una postcondición para que el largo del arreglo final sea N o N + 1. Como tenemos información de `I` y `N` en el código, será fácil para LH identificar si el largo del arreglo aumentó o no. Podemos ver a continuación la implementación de la función:

```
{-@ type ArrayNON1 a N I = { arr : Array a |
  (I <= N => nodeCount arr == N) &&
  (I == N + 1 => nodeCount arr == N + 1) }
@-}
{-@ update1 :: { n: Nat | n > 0 } -> a ->
  arr : ArrayGE a { n - 1 } ->
  ArrayNON1 a { nodeCount arr } { n }
@-}
update1 :: Int -> a -> Array a -> Array a
update1 _ x Nil = singleton x
update1 1 x t@(Node v l r) = makeT x l r
update1 n x t@(Node v l r)
  | even n = makeT v (update1 (div n 2) x l) r
  | otherwise = makeT v l (update1 (div (n - 1) 2) x r)
{-@ reflect update1 @-}
```

Un problema presente en los AVLs y en los árboles balanceados en general es que cuando aumentan su tamaño suelen realizar operaciones de rebalanceo. En este caso, como agregamos un elemento al final del arreglo, no es necesario realizar dicha operación, ya que nunca agregar un elemento rompería el balanceo del árbol.

Por último cabe destacar que, similar a `lookup1` y `lookup` `update1` tiene su función `update` la cual le agrega un 1 al indice para poder adecuarse mejor a los arreglos.

5.2.4. Largo

La función largo nos indica la cantidad de elementos que tiene un arreglo. Como ya tenemos una forma de calcular este valor, esta función es simplemente un alias de nodeCount.

```
len :: Array a -> Int
len arr = nodeCount arr
```

5.2.5. Transformación a lista

Para transformar un árbol en una lista, es necesario que los elementos mantengan en la lista el índice que tenían en el árbol. Es necesario entonces recorrer el árbol en su totalidad intercalando elementos de sus ramas izquierda y derecha. De esta forma, el elemento *n-ésimo* del árbol mantendrá el mismo índice

dentro de la lista. Para lograrlo, implementaremos la función `list`. Esta función une el nodo con las ramas ya transformadas en listas, de manera que el nodo se convierte en el primer elemento de la lista y el resto se obtiene mediante el intercalado de los elementos pertenecientes a las listas obtenidas de las ramas. Para llevar a cabo este intercalado tendremos que implementar la función `splice`, una implementación inicial podría ser la siguiente:

```
splice :: [a] -> [a] -> [a]
splice xs     []     = xs
splice []     ys     = ys
splice (x:xs) (y:ys) = x : y : splice xs ys
```

Sin embargo, en este caso nos interesa probar en secciones futuras algunas propiedades sobre la función `list` por lo que `splice` deberá pertenecer a la lógica de refinado.

Se realizó una primera implementación de esta función, fue utilizando la directiva `measure`, en este caso es necesario utilizar un par para poder mover las dos listas a un único parámetro de entrada.

```
{-@ splice :: p :([a], [a]) ->
  cezs : { [a] | len zs == len (fst p) + len (snd p) }@-}
splice :: ([a], [a]) -> [a]
splice (xs, ys)
  | notEmptyL xs = (head xs)  : splice (ys, (tail xs))
  | otherwise = ys
{-@ measure splice @-}
```

Una de las propiedades que nos interesa verificar es que la lista generada tenga la misma cantidad de elementos que el árbol. Para no perder información acerca del largo de la lista, agregamos la postcondición que iguala el largo de la lista final con la suma de las dos sublistas. De esta manera, podremos chequear que la cantidad de nodos del árbol es igual a la cantidad de elementos de la lista resultante de la operación `splice`.

Como pudimos ver, el `measure` nos requirió cambios sobre la firma de la función, obligándonos a utilizar una estructura auxiliar para satisfacer el requerimiento de un único parámetro de entrada. Si en vez de `measure` utilizamos `reflect` podremos simplificar de forma considerable la función, permitiendo el uso de múltiples parámetros y obteniendo el mismo resultado.

```
{-@ splice :: xs: [a] -> ys: [a] ->
  zs : { [a] | len zs == len xs + len ys } @-}
{-@ reflect splice @-}
splice :: [a] -> [a] -> [a]
splice (x:xs) ys = x : splice ys (xs)
splice [] ys = ys
```

El resultado obtenido se asemeja mucho a la función originalmente propuesta. El caso de la función `splice` proporciona un claro ejemplo de cuándo es más conveniente emplear `reflect` en lugar de `measure`. Si continuáramos utilizando `measures` para las pruebas, estas se volverían más tediosas debido a la manipulación constante de la estructura del par. Una vez que la función splice está implementada, podemos proceder a construir la función `list`, la cual toma un arreglo y devuelve una lista del mismo tamaño.

```
{-@ list :: arr : Array a ->
  { xs : [a] | nodeCount arr == len xs } @-}
list :: (Eq a) => Array a -> [a]
list Nil = []
list (Node x l r) = x : splice (list l) (list r)
{-@ reflect list @-}
```

5.2.6. Correctitud funcional

Una vez implementada la estructura, nuestro interés recae en demostrar algunas propiedades inherentes a la misma. En estructuras previas, las funciones incluían pruebas adjuntas; un ejemplo de ello era el caso del insert en AVL, el cual demostraba que la altura del árbol podía ser igual o mayor a la del original. De manera similar, en esta estructura ya hemos realizado un primer chequeo para asegurar la corrección funcional de las operaciones, añadiendo como postcondición la verificación de la cantidad de nodos en el arreglo (`nodeCount arr`). Esta verificación resulta sencilla de ejecutar, y LH se encarga en su mayoría de demostrar esta propiedad.

No obstante, que las estructuras tengan la longitud adecuada no garantiza su correcta construcción. Por lo tanto, mediante razonamiento ecuacional, demostraremos algunas propiedades que nos permitirán asegurar la validez de la construcción de los árboles y la implementación precisa de las funciones. Las propiedades demostradas a continuación son el equivalente en LH de las demostradas en Isabelle (Nipkow y Sewell, 2020).

Igualdad de lookup

Mediante la operación `list` podemos transformar un arreglo a una lista de largos iguales, pero para asegurarnos que los elementos son *mapeados* con sus índices de forma correcta es necesario chequear que para todo índice el resultado de la operación `lookup` es igual al análogo realizado en la lista.

Es así que buscaremos demostrar el siguiente teorema:

```
list_array_equality :: arr : { Array a | nodeCount arr > 0 } ->
   n : { Nat | n < nodeCount arr } ->
   { lookup n arr == lookupL n (list arr) }
```

Donde `lookupL` será la función `lookup` implementada para las listas con el agregado de que el largo de la lista siempre sea mayor al índice indicado.

Para demostrarlo necesitaremos realizar inducción sobre un árbol sin embargo estos avanzan dentro de su estructura de forma distinta a las listas. Esto presenta un problema, ya que el paso n + 1 en un árbol significa tomar un camino en un subárbol, mientras que para las listas esto es avanzar un elemento dentro de la misma. Es así que, antes de probar el teorema anterior, necesitaremos probar la siguiente propiedad:

```
aux :: [a] -> [a] -> Int -> [a]
aux xs ys n
  | even n = xs
  | otherwise = ys
{-@ inline aux @-}

{-@ lookup_equivalence_list_tree :: (Eq a) =>
  { xs : [a] | len xs > 0 } ->
  ys : { [a] | len ys <= len xs && len xs <= len ys + 1 } ->
  n : { Nat | n < len xs + len ys } ->
  {lookupL n (splice xs ys) == lookupL (div n 2) (aux xs ys n)}
@-}
{-@ ple lookup_equivalence_list_tree @-}
lookup_equivalence_list_tree ::
  (Eq a) => [a] -> [a] -> Int -> Proof
lookup_equivalence_list_tree (x:xs) ys 0 =
  lookupL 0 (splice (x:xs) ys)
  == lookupL (div 0 2) (aux (x:xs) ys 0)
  === lookupL 0 (x: splice ys xs) == lookupL 0 (x:xs)
  === x == x *** QED
lookup_equivalence_list_tree (x:xs) ys n =
  lookup_equivalence_list_tree ys xs (n - 1)
```

Este lema prueba que avanzar dentro de una lista con un índice, es lo mismo que avanzar dentro de una de las sublistas que la descomponen en la función `splice` con el índice dividido entre dos. Nosotros proveemos la demostración del paso base, sin embargo, el *SMT-Solver* se encarga de demostrar el paso inductivo. Esto nos permitirá aplicar inducción en la demostración inicial, ya que podremos movernos a la par entre la lista y los árboles. A continuación el planteo de la demostración utilizando list_tree_lookup_equallity:

```haskell
{-@ list_array_equality :: arr : { Array a | nodeCount arr > 0 }
↪  ->
  n : { Nat | n < nodeCount arr } ->
  { lookup n arr == lookupL n (list arr) } @-}
list_array_equality :: (Eq a) => Array a -> Int -> Proof
list_array_equality  arr@(Node v l r) 0 =
  lookup 0 arr == lookupL 0 (list arr)
  === v == lookupL 0 (v : (splice (list l ) (list r)))
  === v == v *** QED
list_array_equality arr@(Node v l r) n
  | even n = lookup n arr == lookupL n (list arr)
    === lookup1 (n + 1) (arr) ==
        lookupL n (v : (splice (list l) (list r)))
    === lookup1 (div n 2) r ==
        lookupL (n - 1) (splice (list l) (list r))
        ? lookup_equivalence_list_tree (list l) (list r) (n - 1)
    === lookup (div (n -1) 2) r ==
        lookupL (div (n - 1) 2) (list r)
        ? list_array_equality r (div (n - 1) 2)
    *** QED
  | otherwise = lookup n arr == lookupL n (list arr)
    === lookup1 (n + 1) arr ==
        lookupL n (v : (splice (list l) (list r)))
    === lookup1 (div (n + 1) 2) l ==
        lookupL (n - 1) (splice (list l) (list r))
        ? lookup_equivalence_list_tree (list l) (list r) (n -1)
    === lookup (div (n - 1) 2) l ==
        lookupL (div (n - 1) 2) (list l)
        ? list_array_equality l (div (n - 1) 2)
  *** QED
```

Como vemos en esta demostración, tuvimos que proveer un caso base, y le tuvimos que mostrar al solver cómo avanzar para llegar al siguiente paso inductivo. Una vez realizado eso, el mismo chequea el resto y queda demostrado que las funciones `lookup` retornan el mismo resultado.

Es relevante mencionar que la incorporación de estas demostraciones en el código incrementa significativamente el tiempo de compilación. El proceso, que inicialmente tardaba 6.9 segundos, ahora requiere 1 minuto y 29 segundos, lo que representa un aumento de casi 36 veces, equivalente a un 1188.41 %.

Además, observamos un aumento considerable en el tamaño de los archivos generados por LH tras agregar la bandera. El ejemplo más extremo es el archivo .smt2 que contiene las instrucciones utilizadas por el SMT-solver para chequear las pruebas. Este archivo pasa de 16 MB a 332 MB, reflejando un incremento del 1975 %.

Una explicación de este crecimiento del tiempo y espacio podría ser que el

SMT-solver demuestra de forma automática `list_array_equality`. En ciertos casos con muchos teoremas probados de forma automática, LH llegó a estar corriendo 15 minutos sin terminación. Esto indica que, si bien son muy útiles, las pruebas automáticas no vienen sin costos. Podríamos intentar dar una demostración con razonamiento ecuacional en vez de dejar que el *solver* lo haga de forma automática y verificar si los atributos cambiaron. Sin embargo, esto no se contempló para este trabajo y queda como un posible trabajo a futuro.

5.3. Arreglos flexibles

Los arreglos flexibles son aquellos que tienen operaciones para agregar y quitar elementos del arreglo tanto al inicio como al final. En esta sección, explicaremos la implementación de los mismos basándonos en la estructura que construimos anteriormente. Agregaremos las siguientes funciones a la interfaz construida en la sección anterior:

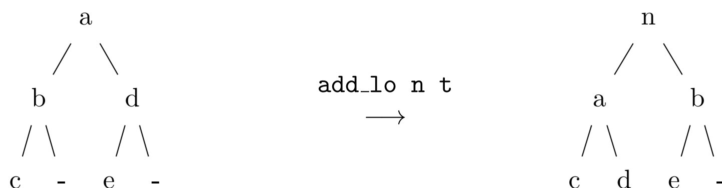
```
add_lo :: a -> Array a -> Array a
del_lo :: Array a -> Array a
add_hi :: a -> Array a -> Array a
del_hi :: Array a -> Array a
```

Cada una de estas funciones realizará la siguiente tarea:

- `add_lo`: Agrega un elemento al inicio del arreglo
- `del_lo`: Elimina el primer elemento del arreglo
- `add_hi`: Agrega un elemento al final del arreglo
- `del_hi`: Elimina un elemento al final del arreglo

5.3.1. Agregar al inicio

Agregar un elemento al inicio del arreglo significa modificar las ramas del árbol para acomodar el nuevo nodo padre. Un ejemplo de agregar un elemento n a un árbol t puede ser el siguiente:



Agregar un elemento `n` al inicio del arreglo `t`

Para implementar la función *add_lo* simplemente tendremos que insertar el elemento nuevo en el inicio del árbol y luego propagar el cambio intercambiando los nodos izquierdos por los derechos. El código de la inserción es el siguiente:

```
{-@ add_lo :: (Eq a) => x : a -> arr : Array a ->
  ArrayN a { nodeCount arr + 1 } @-}
{-@ reflect add_lo @-}
add_lo :: (Eq a) => a -> Array a -> Array a
add_lo x Nil = singleton x
add_lo x (Node v l r) = makeT x (add_lo v r) l
```

Cabe destacar que en este caso el resultado final es un arreglo con un elemento más y es una postcondición que podremos usar luego.

5.3.2. Borrar el primer elemento

Análogo a la anterior: esta operación consta de eliminar el nodo raíz y luego propagar el cambio a todo el árbol intercambiando los elementos. Para esto se necesitará una función auxiliar `merge` que combine dos árboles de la forma indicada previamente. En este caso será necesario agregar algunas precondiciones a la función porque los árboles a `mergear` deben cumplir la condición de balanceo.

```
{-@ merge :: arr1 : Array a ->
  { arr2 : Array a | nodeCount arr2 == nodeCount arr1 ||
                     nodeCount arr1 == nodeCount arr2 + 1 } ->
  ArrayN a { nodeCount arr1 + nodeCount arr2 } @-}
{-@ reflect merge @-}
merge :: Array a -> Array a -> Array a
merge Nil r = r
merge (Node v l r) rr = Node v rr (merge l r)
```

Luego la función de borrado llama a `merge`:

```
{-@ del_lo :: arr : Array a ->
 { arr2 : Array a | nodeCount arr > 0 =>
                    nodeCount arr2 == nodeCount arr - 1  } @-}
{-@ reflect del_lo @-}
del_lo :: Array a -> Array a
del_lo Nil = Nil
del_lo (Node _ l r) = merge l r
```

En este caso, la postcondición necesita de un condicional, ya que `nodeCount Nil` es 0 siempre.

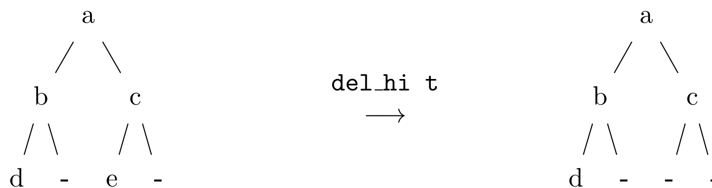
5.3.3. Agregar al final

En secciones previas ya habíamos construido una función que nos permite agregar un elemento al final del árbol. Simplemente, llamaremos a la función `update1` y actualizaremos el índice equivalente a la cantidad de nodos más 1:

```
{-@ add_hi :: a -> arr : Array a ->
  ArrayN a { nodeCount arr + 1 }
@-}
add_hi :: (Eq a) => a -> Array a -> Array a
add_hi a arr = update1 (nodeCount arr + 1) a arr
```

5.3.4. Borrar el último elemento

Eliminar el último elemento significa eliminar la hoja más a la derecha del árbol. Si volvemos al ejemplo de los árboles:



Eliminar el elemento final del arreglo

Para esto, debemos recorrer todo el árbol hasta el último nodo y remplazarlo por `Nil` en vez de un `singleton`. Para indicar que el índice es el final, restringiremos el arreglo para que la cantidad de nodos coincida con el índice ingresado. Luego bastará con llamar a la función con la cantidad de nodos del árbol.

```
{-@ del_hi :: n : Nat -> ArrayN a n ->
{ a : Array a | n > 0 => nodeCount a == n - 1  } @-}
del_hi :: Int -> Array a -> Array a
del_hi _ Nil = Nil
del_hi n (Node v l r)
  | n == 1 = Nil
  | not (even n) = makeT v l (del_hi (div n 2) r)
  | otherwise = makeT v (del_hi (div n 2) l) r
```

5.4. Operaciones eficientes

En secciones anteriores nos concentramos en implementar las operaciones básicas de las estructuras, sin embargo existen formas de mejorar la eficiencia de las mismas. En esta sección nos enfocaremos en implementar de manera eficiente alguna de las operaciones vistas en secciones anteriores sin perder de vista la corrección funcional que nos brinda LH. Para ello nos basamos en dos trabajos sobre BraunTrees: uno desarrollado en Isabelle (Nipkow y Sewell, 2020) y otro de Okasaki (Okasaki, 1997) sobre la misma estructura.

5.4.1. Tamaño

En nuestra función nodeCount original recorríamos todo el árbol contando cada vez que pasábamos por un constructor `Node`. En este caso, es posible hacer uso de la propiedad de balanceo para mejorar la eficiencia del conteo. Para esto, la idea es descartar una rama en la recorrida, dependiendo de la cantidad de nodos que tiene la otra. Es así que escribimos las siguientes funciones:

```
{-@ diff :: n: Nat ->
  arr : { Array a | nodeCount arr == n ||
                    nodeCount arr == n + 1 } ->
  { nf : Nat | nf == nodeCount arr - n } @-}
diff :: Int -> Array a -> Int
diff _ Nil = 0
diff n (Node _ l r)
  | n == 0 = 1
  | even n  = diff ((div n 2) -1) r
  | otherwise = diff (div n 2) l

{-@ size_fast :: arr : Array a ->
  { n : Nat | n == nodeCount arr }
@-}
size_fast :: Array a -> Int
size_fast Nil = 0
size_fast (Node _ l r) = 1 + 2 * n + diff n l
  where
    n = size_fast r
```

La función `diff` calcula la diferencia entre la cantidad de nodos de dos ramas. Este cálculo se lleva a cabo recorriendo una de las ramas con la cantidad de nodos de la otra. Dependiendo de la paridad de este valor, determinamos el camino a recorrer, lo que nos permite evitar recorrer una de las ramas en cada iteración.

Por otro lado, `size_fast` define el cálculo general multiplicando por dos la cantidad de nodos del árbol derecho, que, por definición de BraunTrees, siempre tiene la misma cantidad de nodos que el izquierdo o uno menos. Luego, se

agrega 1 por el nodo padre y se suma la posible diferencia entre la cantidad de nodos de los árboles derecho e izquierdo. Este conteo se realiza en un orden de O(`log2(nodeCount(t))`).

Es importante destacar que como postcondición, la función `size_fast` tiene efectivamente el mismo valor que la operación `nodeCount`. Esto es probado por LH sin necesidad de agregar ningún tipo de demostración

5.4.2. Altura

En (Nipkow y Sewell, 2020) se presenta una forma de calcular la altura más eficiente. Dada la prioridad que tiene el árbol izquierdo en la estructura, si calculamos la altura del mismo, entonces el derecho siempre tendrá la misma o un punto menos. Por ende, podríamos tener una operación de cálculo de altura de orden O(`log2(nodeCount(t))`). La función podemos verla a continuación:

```
{-@ lh :: arr : Array a -> l : { Nat | l == height arr }@-}
lh :: Array a -> Int
lh Nil = 0
lh (Node _ l _) = 1 + lh l
```

Para poder demostrar la condición `l == height arr` es necesario relacionar la altura con la cantidad de nodos mediante la siguiente propiedad:

$$log2(nodeCount(t)) = height(t) \tag{5.1}$$

Esta propiedad, sumada a las restricciones estructurales que poseen los BraunTrees haría muy eficiente el cálculo de la altura. Sin embargo, esto no fue posible demostrarlo en LH y quedó como trabajo a futuro. La principal dificultad es integrar la noción de logaritmo a la lógica de refinado.

5.4.3. Generar un arreglo de largo n

Muchas veces es útil generar arreglos de cierto largo que contengan un valor por defecto. Una forma de realizar esta operación sería agregar un nodo al inicio o al final del arreglo n veces desde el arreglo vacío. Esto tendría un orden *O(nlog2(n))* en ambos casos. Sin embargo, es posible hacerlo de forma más eficiente. Para esto utilizaremos un par para construir un arreglo de largo impar y otro de largo par. Luego, dependiendo del largo del arreglo, duplicaremos uno de los dos árboles generados para incrementar la cantidad de nodos de forma más eficiente. La operación se repite hasta alcanzar la cantidad de nodos deseados en orden *O(log2(n))* y se implementa de la siguiente forma:

```
{-@ braun2_of :: a -> n : Nat -> (ArrayN a {n + 1}, ArrayN a n)
 @-}
braun2_of :: (Eq a) => a -> Int -> (Array a, Array a)
braun2_of x 0 = (singleton x, Nil)
braun2_of x n
  | even n = let (s,t) = braun2_of x (div (n-2) 2) in
                         (makeT x s s, makeT x s t)
  | otherwise = let (s,t) = braun2_of x (div (n-1) 2) in
                         (makeT x s t, makeT x t t)

{-@ braun_of :: a -> n: Nat -> ArrayN a n @-}
braun_of :: (Eq a) => a -> Int -> Array a
braun_of x n = snd (braun2_of x n)
```

Como podemos ver, LH es capaz de unificar las postcondiciones de los pares, obteniendo así el resultado esperado sin la necesidad de realizar grandes pruebas.

5.4.4. Transformación de una lista a un arreglo

Para transformar una lista a un arreglo, Okasaki (Okasaki, 1997) presenta la siguiente función, como mejora a la anterior forma de crear arreglos, a través de listas en orden $O(n)$ donde `n` es la cantidad de nodos del árbol:

```
rows k [] = []
rows k xs = []
rows k xs =  (k, take k xs) : rows (2 * k) (drop k xs)

build (k, xs) ts = zipWith3 makeNode xs ts1 ts2
  where (ts1, ts2 ) = split k (ts ++ repeat ⟨⟩)

makeNode x s t = ⟨x, s, t⟩
makeArray = head o foldr build [⟨⟩] o rows 1
```

La estrategia implica el uso de cuatro funciones. La primera, `rows`, divide una lista en pares: un entero k y una lista de longitud k. Su propósito es generar una lista con los elementos que conformarán cada nivel del árbol. Por eso, aumenta la cantidad de elementos en cada lista en 2 veces k.

La función `build` une los nodos de un nivel con una lista de árboles compatibles para ser BraunTrees. Retorna una lista de arreglos compatibles de longitud k / 2.

Finalmente, *makeArray* combina todas estas funciones. Espera recibir una lista, la divide en niveles de elementos utilizando la función `rows`, y luego va uniendo cada nivel en un acumulador mediante `build`. En el último nivel, esta lista tendrá un único elemento, que será el arreglo final. Por eso, se extrae llamando al primer elemento de la lista.

Rows

Comenzamos reescribiendo la función `rows` de forma que podamos lograr resultados similares con LH. Lo importante a notar es que para crear un Braun-Tree necesitamos que la cantidad de nodos de los árboles sea compatible. Por lo que, a lo largo de la implementación de estas funciones, la cantidad de nodos del arreglo será un dato de vital importancia para que LH pueda verificar la correcta construcción de los árboles. Okasaki utiliza la función `rows` para generar una lista de pares que tienen la cantidad de elementos de cada nivel del árbol menos el último que tiene el resto. Si intentáramos usar la estructura de un par, terminaríamos con una casuística bastante compleja de representar en las funciones, es así que decidimos crear una estructura llamada `IRow` la que cumplirá la función de la lista de pares.

```haskell
data IRow a where
  Last :: (Eq a) => Int -> [a] -> IRow a
  IRow :: (Eq a) => Int -> [a] -> IRow a -> IRow a

{-@
data IRow a where
  Last :: (Eq a) => { n: Nat | n > 0} ->
    { l: [a] | len l > 0 && len l <= n } -> IRow a
  IRow :: (Eq a) => { n: Nat | n > 0} ->
    { l : [a] | len l == n } ->
    {r : IRow a | size r == 2 * n } -> IRow a
@-}
```

Como podemos ver tenemos dos constructores:

- El constructor `Last` indica el fin de la lista de pares, la cual posee la cantidad de elementos del nivel n y una lista con cantidad de elementos menor o igual a `n`.
- El constructor `IRow` toma la cantidad de elementos del nivel, la lista con la misma cantidad de elementos y los siguientes niveles. La función `size` es un alias del atributo n.

Por último, para construir la función `row` necesitamos una forma de unificar que el largo de la lista original es igual a la cantidad de elementos contenidos en el `IRow`. Es así que definimos la siguiente función:

```haskell
countRowNodes :: (Eq a) => IRow a -> Int
countRowNodes (Last _ xs) = length xs
countRowNodes (IRow n xs next) = n + countRowNodes next
{-@ measure countRowNodes @-}
```

Que cuenta la cantidad de nodos de cada lista, como podemos ver en el caso de el constructor `IRow` no es necesario llamar a la función `length`. Una vez

definidos todos nuestros componentes podemos escribir la funcion `rows` de la siguiente forma:

```
{-@ rows :: (Eq a) => { n : Nat | n > 0 } ->
  { xs:[a] | len xs > 0 } ->
  { ir : IRow a | size ir == n && countRowNodes ir == len xs }
@-}
{-@ reflect rows @-}
rows ::(Eq a) => Int -> [a] -> IRow a
rows k xs
  | notEmptyL d = IRow k t (rows (2 * k) d)
  | otherwise = Last k xs
  where
  (t,d) = split k xs
```

Como podemos observar, tenemos como postcondición que la cantidad de elementos de la lista es n y la cantidad de elementos almacenados en `IRow` es la misma que la lista.

Build

La función `build` presenta un problema similar al que tuvimos que resolver en `rows`. Okasaki utiliza una lista de árboles, pero sus funciones no tienen ninguna restricción estructural, sin embargo, nosotros sí. Debido a esto utilizaremos otra estructura auxiliar, `IArray` para poder generar listas de árboles que tengan una cantidad de nodos *nc* y a partir del índice i la cantidad de nodos de los árboles será nc - 1. Es así que definimos la estructura `IArray` de la siguiente forma:

```
data IArray a where
  IArray :: (Eq a) => Int -> Int ->
    [Array a] -> [Array a] -> Int -> IArray a

{-@
data IArray a where
  IArray :: (Eq a) => { nc : Nat | nc > 0 } ->
   i : { Nat | i > 0 } ->
   xs : ListN (ArrayN a nc) i ->
   ys : List (ArrayN a { nc - 1 }) ->
   {l : Nat | l == len xs + len ys} -> IArray a
@-}
```

Como podemos ver, `IArray` contiene dos atributos nc e i que representan la cantidad de nodos y el índice a partir del cual la cantidad baja. Dos listas que contienen los arreglos con la cantidad de nodos ya mencionados y un atributo

largo para poder calcular la cantidad de elementos fácilmente. Una vez construida esta estructura podremos entonces plantear la función `build` de la siguiente manera:

```
{-@ build :: (Eq a) => t: IRow a ->
 { iarr : IArray a | sizeIArray iarr == size t
    && countNodesArr iarr == countRowNodes t }
 @-}
build :: (Eq a) => IRow a -> IArray a
build (Last k xs) =
    IArray 1 len2 (map singleton xs) (compleate rest (Nil) []) k
  where
    len2 = length2 xs
    rest = (k - len2)
build (IRow k xs next)
  | index > k = buildkg k iarr xs
  | index == k = buildke k iarr xs
  | index < k = buildkl k iarr xs
  where
    iarr@(IArray nc index narrayl narrayr _) = build next
```

La función `build` se divide en dos casos principales. El primero ocurre cuando alcanzamos el último elemento del IRow; en este escenario, convertimos la lista de elementos en árboles de un solo nodo. Posteriormente, completamos la otra lista con nodos vacíos para asegurar la cantidad necesaria de nodos en el nivel más bajo del árbol.

El segundo caso corresponde a un elemento inicial o intermedio del `IRow`, lo que nos lleva a construir el `IArray` del elemento siguiente. Una vez que este `IArray` está construido, comparamos el índice que indica la cantidad de árboles con n nodos con la cantidad de elementos k necesarios para el nivel. Según esta comparación, realizamos diferentes operaciones de construcción del `IArray`. En la próxima sección, detallaremos estos casos uno por uno. Antes, introduciremos algunas nociones necesarias para la construcción de estos elementos:

Subtipos de IArray

Para abreviar algunas de las pre y postcondiciones de la próxima sección, se construyeron subtipos de los `IArray`. Estos subtipos utilizan las funciones `indx` y `getK`. Ambas funciones son `measures` que nos permiten el acceso a un atributo del GADT y, por ende, nos permiten la restricción de los mismos como condiciones. Es así que definimos al `IArray` con el índice mayor, igual o menor a k y con la cantidad de elementos suficientes para completar el nivel. Además, definimos el `IArrayNKN` el cual establece que el tamaño del arreglo es igual a NK lo que quiere decir que la cantidad de nodos por nivel y la cantidad de nodos es igual a N.

```
{-@
type IArrayGK a NK = {
  arr : IArray a | indx arr > NK && getK iarr == 2 * NK }
type IArrayEK a NK = {
  arr : IArray a | indx arr == NK && getK iarr == 2 * NK }
type IArrayLK a NK = {
  arr : IArray a | indx arr < NK && getK iarr == 2 * NK }
type IArrayNKN a NK N = {
  arr : IArray a |
  sizeIArray arr == NK && countNodesArr arr == N }
@-}
```

Remplazo del zipWith3

Como pudimos ver en el código, Okasaki utiliza el `zipWith3` para unir los elementos de las listas. Esto no es posible en LH, ya que en este caso tenemos que unir árboles de distinta cantidad de nodos y al utilizar el `zipWith3` estándar, perdemos esta noción. Es así que implementamos una función que recibe dos listas de árboles y las une en una sola:

```
{-@ type SOL N = { n : Nat | n == N || n == N - 1 } @-}
{-@ makeT3 :: (Eq a) => nc1 : Nat ->
  nc2 : SOL nc1 -> xs : [a] ->
  ys : ListN (ArrayN a nc1) (len xs) ->
  zs : ListN (ArrayN a nc2) (len xs) ->
   ListN (ArrayN a { 1 + nc1 + nc2 }) (len xs) @-}
makeT3 :: (Eq a) => Int -> Int -> [a] -> [Array a] ->
  [Array a] -> [Array a]
makeT3 nc1 nc2 [] [] [] = []
makeT3 nc1 nc2 (x:xs) (y:ys) (z:zs) =
  makeT x y z : (makeT3 nc1 nc2 xs ys zs)
```

Caso index mayor a la cantidad de árboles

Una vez definidas todas las funciones y tipos auxiliares que necesitábamos, podemos construir las distintas formas de `build`. En este caso, el `index` es mayor a la cantidad de árboles necesarios por nivel. Esto significa que una parte de los árboles construidos tendrán altura 2 multiplicado la cantidad de nodos + 1 y otros solo 2 por cantidad de nodos. La idea será calcular cuál es la cantidad de árboles y generar un `IArray` que tenga como índice ese valor, como podemos ver a continuación:

```
{-@ buildkg :: (Eq a) => nk : Nat ->
 iarr : IArrayGK a nk  ->
 rs: { [a] | len rs == nk } ->
 IArrayNKN a { nk } { len rs + countNodesArr iarr }
@-}
buildkg :: (Eq a) => Int -> IArray a -> [a] -> IArray a
buildkg k (IArray nc index ys zs _) xs =
  IArray (1 + 2* nc) splitindex narr n1arr k
  where
    narr = (makeT3 nc nc xs1 an3 an2)
    n1arr = (makeT3 nc (nc - 1) xs2 an4 zs)
    (xs1, xs2) = split splitindex xs
    (an, an2) = split k ys
    (an3, an4) = split splitindex an
    splitindex = index - k
```

Caso index igual a la cantidad de árboles

En el caso de que el índice es igual a la cantidad de elementos del nivel, construiremos árboles con cantidad de nodos `2 * nc`, ya que uniremos los árboles con cantidad de nodos `nc` y `nc -1` con un elemento. De esa forma, la función `buildke` se implementa de la siguiente manera:

```
{-@ buildke :: (Eq a) => nk: Nat  -> iarr : IArrayEK a nk ->
rs:{ [a] | len rs == nk } ->
IArrayNKN a { nk } { len rs + countNodesArr iarr }  @-}
{-@ reflect buildke @-}
buildke :: (Eq a) => Int -> IArray a -> [a] -> IArray a
buildke k (IArray nc index ys zs _) xs =
  IArray (2 * nc) k (makeT3 nc (nc -1) xs ys zs) [] k
```

Caso index menor a la cantidad de árboles

Para este caso, similar al inicial, necesitaremos separar entre los árboles que tendrán cantidad de elementos `2 * nc` y `2 * nc -1`

```
{-@ buildkl :: (Eq a) => nk: Nat ->
  iarr: IArrayLK a nk ->
  rs:{ [a] | len rs == nk } ->
  IArrayNKN a { nk } { len rs + countNodesArr iarr }
@-}
buildkl :: (Eq a) => Int -> IArray a -> [a] -> IArray a
buildkl k (IArray nc index ys zs _) xs =
  IArray (2* nc) index narr n1arr k
  where
    narr = (makeT3 nc (nc - 1) xs1 ys an3)
    n1arr = (makeT3 (nc - 1) (nc - 1) xs2 an an4)
    (xs1, xs2) = split index xs
    (an, an2) = split (k - index) zs
    (an3, an4) = split index an2
```

Implementación de makeArray

Por último, uniremos todo de forma muy similar a cómo lo hace Okasaki por lo que llamaremos a la función `makeArray` de la siguiente manera:

```
{-@ makeArray :: (Eq a) => xs: [a] -> arr : ArrayN a { len xs }
 @-}
makeArray ::(Eq a) => [a] -> Array a
makeArray [] = Nil
makeArray xs = head arr
  where
    (IArray _ i arr arr2 k) = (build (rows 1 xs))
```

Como podemos ver, le agregamos como postcondición que la cantidad de elementos del arreglo sea igual al largo de la lista inicial y LH es capaz de chequearla de manera correcta. De esta forma logramos implementar el algoritmo planteado por Okasaki con LH. En este caso, resta probar la correctitud funcional de la función igualando la función `lookup`; esto no entró dentro del proyecto y es uno de los puntos a trabajar en el futuro.

Capítulo 6

Conclusiones

Durante esta investigación, exploramos diferentes estructuras para comprender las fortalezas y debilidades de LH. Desarrollamos cuatro estructuras distintas: ABB Internalista, Externalista, AVL y BraunTrees. Cada una con diferentes invariantes, basadas en una misma estructura base, el árbol binario. A través de este desarrollo podemos concluir lo siguiente:

Los ABB nos permitieron adentramos en conceptos básicos de la herramienta, empleando `measures` para promover funciones a la lógica de refinamiento. Comenzamos con un enfoque externalista, utilizando numerosas funciones auxiliares para asegurar las postcondiciones necesarias en las funciones `insert` y `delete`. Posteriormente, al implementar la versión internalista, pudimos eliminar estas funciones, empleando la estructura misma para hacer cumplir dichas condiciones. La comparación entre estas dos estructuras destacó la notable reducción de código que significó la implementación internalista. Además, en el enfoque internalista, el cuerpo de las funciones resultó similar a la implementación más básica de los ABB, lo que es más familiar y legible para el usuario.

Seguimos con la implementación de árboles AVL, donde comenzamos a visualizar algunas limitaciones de eficiencia al verificar la condición de balanceo en todos los niveles. Sin embargo, LH nos permitió resolver estos problemas de manera sencilla mediante el uso de `measures` y la adición de condiciones a los GADT's. Durante el desarrollo, también observamos las primeras limitaciones del SMT-solver, ya que no pudo resolver automáticamente todas las condiciones que agregamos a nuestras funciones. Por tanto, la posibilidad de deducir los chequeos agregados depende de la precisión con la que establecemos las pre y postcondiciones de nuestras funciones. Si bien el resultado fue exitoso, el hallar la solución a este tipo de problemas no fue trivial.

Los BraunTrees nos permitieron implementar condiciones basadas en índices y su paridad, donde pudimos ver la importancia que tiene la condición `even` dentro de la implementación. Experimentamos con las directivas `reflect` y `measure`, observando las ventajas de la primera sobre la segunda. Además, exploramos el razonamiento ecuacional y demostramos propiedades cuya incorporación directa en las funciones habría aumentado su complejidad. Sin embargo,

esta aproximación no fue sin costos; el uso de pruebas automáticas incrementó considerablemente los tiempos de compilación y el tamaño de los archivos. Implementamos operaciones eficientes para los BraunTree y notamos otra limitación de LH: su incapacidad para razonar sobre propiedades logarítmicas. Al implementar la función `makeArray` para lograr una implementación más eficiente, aumentamos su complejidad, demostrando que no todas las condiciones son fáciles de probar y que la adición de propiedades de LH conlleva compromisos al implementar funciones.

En conclusión, LH es una herramienta útil para demostrar ciertas propiedades del lenguaje que no serían posibles de demostrar con Haskell convencional. Sin embargo, tiene características específicas que deben considerarse al desarrollar código, y la forma y precisión con la que planteamos los problemas resulta crucial. Requiere una familiaridad con los métodos de resolución de teoremas propios del SMT-solver que utilizamos para lograr soluciones más eficientes. Por último, aunque las herramientas automáticas son generalmente útiles, en muchos casos pueden generar costos computacionales significativos, llegando incluso a la no terminación del proceso.

Trabajo Futuro

Entre los trabajos futuros, se encuentra la mejora en el rendimiento de la prueba realizada en la subsección 5.2.6. Implementar una prueba que no utilice técnicas automáticas podría reducir el tiempo de ejecución de las demostraciones. Además, se propone explorar el impacto de las últimas versiones de LH en las pruebas mencionadas, dado que estas incorporan un nuevo operador y optimizaciones que podrían mejorar su eficiencia. Respecto a los BraunTrees desarrollar una versión internalista podría dar resultados diferentes en algunos casos y es algo que se podría explorar a futuro. También se dejó fuera del alcance del trabajo la posibilidad de realizar operaciones con logaritmos, lo que podría lograrse mediante la creación de axiomas o funciones que activen propiedades requeridas como postcondiciones. Además, la completa corrección funcional de `makeArray` no se abordó en este proyecto, quedando la posibilidad de realizarla a través de deducción ecuacional similar al trabajo realizado con `list_array_equality`. Por último, se propone como trabajo futuro comparar estas estructuras en términos de eficiencia, cantidad de líneas de código y tiempos de compilación con sus equivalentes en tipos dependientes.

Referencias

Bertot, Y., y Castéran, P. (2004). *Interactive theorem proving and program development: Coq'art: The calculus of inductive constructions*. Springer.

de León, F. (2024a). *Liquid structures.* https://github.com/f7deleon/liquid_structures/blob/061732b3443848108a8a4607e13e1d4cc3be811f/src/ABBTreesExt.hs. GitHub.

de León, F. (2024b). *Liquid structures.* https://github.com/f7deleon/liquid_structures/blob/061732b3443848108a8a4607e13e1d4cc3be811f/src/AVLTrees.hs. GitHub.

de León, F. (2024c). *Liquid structures.* https://github.com/f7deleon/liquid_structures/blob/061732b3443848108a8a4607e13e1d4cc3be811f/src/BraunTrees.hs. GitHub.

Hoogerwoord, R. R. (1992). A logarithmic implementation of flexible arrays. En R. S. Bird, C. Morgan, y J. Woodcock (Eds.), *Mathematics of program construction, second international conference, oxford, u.k., june 29 - july 3, 1992, proceedings* (Vol. 669, p. 191-207). Springer.

Jhala, R. (2013). *Refinement types 101.* Descargado 2024-05-15, de https://ucsd-progsys.github.io/liquidhaskell/blogposts/2013-01-01-refinement-types-101.lhs/

Jhala, R. (2020a). *How to install liquidhaskell docs.* Descargado 2024-05-15, de https://ucsd-progsys.github.io/liquidhaskell/install/

Jhala, R. (2020b). *Options and pragmas liquidhaskell docs.* Descargado 2024-05-15, de https://ucsd-progsys.github.io/liquidhaskell/options/

Jhala, R., Seidel, E., y Vazou, N. (2023). *Programming with refinement types.* Descargado de https://ucsd-progsys.github.io/liquidhaskell-tutorial/Tutorial_02_Logic.html (Disponible online, Capitulo 2 seccion: Uninterpreted Function)

Nipkow, T., y Sewell, T. (2020). Proof pearl: Braun trees. En J. Blanchette y C. Hritcu (Eds.), *Certified programs and proofs, CPP 2020* (p. 31).

Norell, U. (2007). *Towards a practical programming language based on dependent type theory.* SE-412 96 Göteborg, Sweden.

Okasaki, C. (1997). Three algorithms on braun trees. En (Vol. 7, p. 661–666). Cambridge University Press.

Ranjit Jhala, N. V., Eric Seidel. (s.f.). *Case study: Avl trees.* Descargado 2024-05-15, de http://ucsd-progsys.github.io/liquidhaskell-tutorial/Tutorial_12_Case_Study_AVL.html

Vazou, N., Breitner, J., Kunkel, R., Van Horn, D., y Hutton, G. (2018). Theorem proving for all: equational reasoning in liquid haskell (functional pearl). En *Proceedings of the 11th acm sigplan international symposium on haskell* (p. 132–144). Association for Computing Machinery.

Vazou, N., Rondon, P. M., y Jhala, R. (2013). Abstract refinement types. En M. Felleisen y P. Gardner (Eds.), *Programming languages and systems* (pp. 209–228). Berlin, Heidelberg: Springer Berlin Heidelberg.

Vazou, N., Seidel, E. L., Jhala, R., Vytiniotis, D., y Peyton-Jones, S. (2014). Refinement types for haskell. En *Proceedings of the 19th acm sigplan international conference on functional programming* (p. 269–282). Association for Computing Machinery.

Vazou, N., Tondwalkar, A., Choudhury, V., Scott, R. G., Newton, R. R., Wadler, P., y Jhala, R. (2017, dec). Refinement reflection: complete verification with smt. En (Vol. 2). New York, NY, USA: Association for Computing Machinery.

Apéndice A

Prueba de equivalencia función isABB

En esta sección se presenta una prueba de equivalencia entre la definición formal de los árboles binarios de búsqueda y la función `isABB`.

Empezaremos por enunciar la definición formal de los ABB: Sea x un nodo de un ABB. Si y es un nodo del subárbol izquierdo perteneciente a x, entonces $key(y) < key(x)$. Si y es un nodo perteneciente al subárbol derecho de x entonces $key(x) < key(y)$

Siendo `key` la siguiente función en nuestra representación de los ABB:

```
{-@ key :: (Ord a) => t : { Tree a | notEmpty t } -> a @-}
key :: (Ord a) => Tree a -> a
key (Node k _ _) = k
```

Por motivos prácticos nos referiremos a los arboles que cumplen con la definición formal como `abb(x)`. Entonces pasaremos a demostrar el siguiente condicional:

$$\forall x, isABB(x) \iff abb(x) \tag{A.1}$$

Demostración:

Sea `x = Node v l r`:

$(\Rightarrow)$

Intentaremos demostrar que si `isABB(x)` es verdadero, entonces el arbol cumple la definición formal.

Empecemos estudiando la rama izquierda, en caso de que la misma sea vacía entonces la definición formal se cumple ya que no hay ninguna `key` dentro de la rama izquierda que sea mayor al valor del nodo x. En caso de que la rama no sea vacía, entonces tomamos n un nodo perteneciente a el subárbol l por definición de `isABB`, existe un numero g talque `g = gratest(l)` y se cumple que `gratest`

`l <v`. La función `gratest` nos proporciona el mayor valor dentro del subárbol l por ende se cumple que $\forall n \leq g < v$, por lo que se cumple la definición formal.

En el caso de la rama derecha, si el subárbol es vació entonces la demostración es análoga a la anterior. Si la rama no es vacía entonces tomamos n nodo de la rama r, por definición de $isABB$ sabemos que existe un numero k tal que `k = lowest(r)` y se cumple que `lowest(r) >v`. Por la definición de `lowest` sabemos que retorna el menor de los elementos del arbol por ende se cumple que $n >= k > v$ para cualquier n perteneciente a la rama derecha, por lo que se cumple la definición formal.

Habiendo demostrado que para ambas ramas se cumple la definición formal, concluimos que si el árbol cumple con `isABB` entonces cumple la definición formal.

($\Leftarrow$)
Si x cumple la definición formal tendremos que demostrar que `isABB(x)` es verdadero.

Sea l el subárbol izquierdo de x, existe un conjunto de elementos acotado L que contiene todos los valores de las *key* de l, como es un conjunto de valores acotado tiene máximo y es parte del conjunto. Por definición de l todos los elementos son menores a v por lo tanto su máximo también es menor.

Sea r el subárbol derecho de x,existe un conjunto de elementos acotado R que contiene todos los valores de r, como es un conjunto de valores acotado tiene mínimo y es parte del conjunto. Por definición de r todos los elementos son mayores a v por lo tanto el mínimo del conjunto también lo es.

Ambos arboles l y r cumplen con `isABB` y las condiciones de desigualdad del elemento mayor y menor por lo que `isABB` se evalúa como verdadera.

$\blacksquare$

En conclusión demostramos que la función `isABB` es análoga a la definición formal de los ABB.