Universidad Nacional Autónoma de México

Facultad de Ciencias

Verificación formal de arboles roji-negros

T E S I S

que para obtener el título de:

Licenciado en Ciencias de la Computación

presenta:

David Felipe Hernández Chiapa

Tutores:

Dra. Lourdes del Carmen Gonzalez Huesca

Ciudad Universitaria, Ciudad de México, 2020

Dedicatoria ...

Agradecimientos

Tesis realizada bajo el proyecto PAPIME 102117 “Tópicos en Ciencia de la Computación Teórica”

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Hoy en día en el desarrollo de software existe un conjunto de normas a las cuales se les denomina *buenas prácticas de programación*, las cuales van desde tener una correcta indentación, en particular, elegir si usaremos espacios o tabuladores para realizar este acomodo, la documentación del código, respetar las convenciones del lenguaje que estamos usado y verificar que nuestro programa se comporta de la manera deseada, en específico, se pueden realizar pruebas unitarias.

Las *pruebas unitarias* nos ayudan a saber si un código tiene el comportamiento que buscamos, pero esto soló nos sirve hasta cierto punto; por ejemplo, si tenemos una función que recibe un par de números naturales, para poder verificar que la función es correcta se tendrían que probar todos los casos de entradas, es decir, todas las combinaciones de pares de números naturales que existan, sin embargo, estas combinaciones son infinitas y se necesitaría la misma cantidad de memoria y de tiempo para poder ejecutar una prueba unitaria exhaustiva. Teóricamente esto es posible, pero en la práctica simplemente no contamos con los recursos suficientes.

Siendo así, escribir una prueba unitaria exhaustiva no es factible, en tal caso ¿que podríamos hacer para verificar una implementación?; escribir una prueba unitaria que itere sobre un conjunto representativo de los datos que la función puede recibir como argumentos. Sin embargo, ¿que se puede esperar si la misma prueba unitaria es errónea?, no hay una respuesta clara para esto y la misma industria hoy en día utiliza métodos, como el expuesto anteriormente, para probar código pero ciertamente esto no es suficiente para decidir si el programa es correcto respecto a una especificación.

La única manera en que se puede demostrar que una función o programa es correcto respecto a una especificación es mediante una prueba matemática formal, el problema con este método es que es muy complejo, complicado y a veces tardado para usarse en la industria o en el día a día. A lo largo del tiempo

se ha buscado la manera de hacer este proceso mas amigable al programador, un ejemplo de esto son los lenguajes de programación funcionales, como lo seria Haskell. Este paradigma lleva a los programas a un contexto donde la notación es muy parecida a lo que se usaría en las matemáticas tradicionales, es decir, funciones que van de un conjunto de datos a otro. Damos como ejemplo la función que calcula el factorial de un número, esta la escribiremos tanto en Haskell como en la notación que suele usarse en cursos de matemáticas tradicionales, ver figuras 1.1 y 1.2. Podemos apreciar como las definiciones son casi idénticas, ya que ambas definen los siguientes puntos:

- Ambas definen el tipo de sus variables, o en otras palabras, el conjunto al que las variables pertenecen.
- Ambas definiciones establecen de que tipo de dato toman sus argumentos, es decir, la entrada de la función es un numero natural al igual que el resultado.
- Podemos intercambiar 'f' por 'fac' en cualquiera de las dos definiciones y el significado no cambiaría.

```
fac :: (Integral n) => n -> n
fac 0 = 1
fac n = n * fac (n - 1)
```

Figura 1.1: Función factorial, Haskell.

Sea n $\in \mathbb{N}$ y $f(n)$ la función factorial definida como sigue:
$f : \mathbb{N} \to \mathbb{N}$
$f(0) = 1$
$f(n) = n * f(n-1)$

Figura 1.2: Función factorial.

Se puede notar como las definiciones corresponden casi perfectamente la una con la otra, esto facilita la demostración formal de los programas escritos en estos tipos de lenguajes, sin embargo, estas demostraciones son realizadas de manera tradicional usando lapíz y hojas de papel, no obstante, la única manera en que estemos seguros acerca de la corrección de la desmostración es que otra persona lea, entienda y valide la misma. Todos estos procesos, creación y revisión de la demostración, están hechos por humanos, por lo tanto son susceptibles a errores.

En los últimos años han entrado en desarrollando diversos programas que nos ayudan a solucionar este tipo de problemas, como son los *demostradores automáticos* y los *asistentes de prueba*. Estas herramientas son sistemas o programas que realizan o gestionan demostraciones. Nosotros nos enfocaremos en el uso del segundo; en particular en este trabajo usaremos el asistente interactivo

de pruebas llamado *Coq*[1]. Este nos ayudará guiándonos por la prueba, llevando un control de los casos que nos falten por demostrar y las hipótesis disponibles para cada caso, todo esto sobre programas escritos en el lenguaje funcional del mismo asistente.

Sin embargo, el uso de este asistente genera otro problema, no podemos probar cualquier programa escrito en un lenguaje funcional, primero tenemos que traducir este programa al lenguaje de *Coq* para poder comenzar con las demostraciones. Aquí tenemos dos opciones: traducir a mano y/o adaptar una implementación al lenguaje de *Coq* o utilizar una herramienta que nos ayude a traducir. En este trabajo usaremos la segunda opción, una herramienta llamada *hs-to-coq* [Spector-Zabusky et al., 2017], para probar formalmente la correctud [2] de una estructura de datos como lo son los árboles roji-negros.

Tomando como referencia el trabajo [López Campos, 2015], en donde se realizaron diversas implementaciones de árboles roji-negros usando el lenguaje de programación Haskell, por ejemplo, se desarrollaron implementaciones usando constructores inteligentes y otra implementación mas compleja usando tipos anidados. Siendo este trabajo la principal motivación de elegir esta estructura de datos no trivial para realizar una verificación formal de la implementación de constructores inteligentes.

1.2. Árboles Roji-negros

Los árboles roji-negros son una estructura de datos donde sus operaciones de inserción, eliminación y búsqueda se efectúan en tiempo logarítmico, es decir, la complejidad de estas operaciones es: $O(log(n))$. Los árboles roji-negros son una subclase de los árboles binarios de búsqueda, en los cuales la complejidad de dichas operaciones crece hasta $O(n)$, como si estos fueran una lista simple o doblemente ligada. Esta mejora se obtiene gracias a la introducción de colores a los nodos del árbol (rojo y negro, de ahi rojinegros) y a invariantes relacionados con estos, los cuales describiremos en la siguiente definición.

Defincíon 1.2.1. (Definición de árboles roji-negros) Un árbol binario de búsqueda es un árbol roji-negro si satisface lo siguiente:

1. Todos sus nodos son rojos o negros.
2. El árbol vacío es negro.
3. La raíz es negra [3].
4. Las siguientes invariantes se tienen que cumplir:
    - Un nodo rojo debe tener descendientes negros.

[1] Como en este trabajo no se profundizara sobre el funcionamiento de la herramienta, presentamos la pagina de referencia del mismo: `https://coq.inria.fr/`

[2] Del inglés *correctness*.

[3] Decímos que un árbol es negro o rojo si el nodo de la raiz es de ese color.

- Todos los caminos de la raíz a las hojas deben tener la misma cantidad de nodos negros.
- Todas las hojas del árbol son vacías y de color negro.

Decimos que la altura negra de un nodo es el número de nodos negros que aparecen en el camino de ese nodo a la raíz.

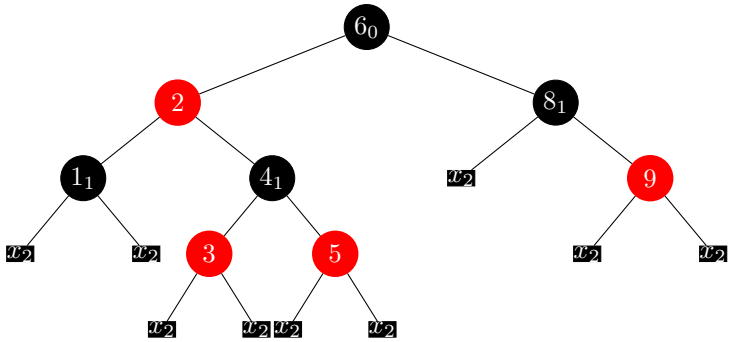

Figura 1.3: Árbol Roji-negro

En la figura 1.3 podemos ver un ejemplo de un árbol roji-negro que respeta la definición que acabamos de presentar; las etiquetas de los nodos representan la información que se puede almacenar en ellos, siendo en este caso números naturales, las etiquetas x en las hojas representan los nodos vacíos y estos son negros, y los subíndices de que aparecen en los nodos negros representan la altura negra del subárbol. Con este ejemplo se ilustra como la cantidad de nodos negros de la raíz a cualquier hoja es siempre la misma.

Nos interesa estudiar este tipo no trivial de *árboles binarios de búsqueda* para poder demostrar la correctud de la implementación funcional en [López Campos, 2015] usando el asistente de pruebas *Coq* y de esta manera poder mostrar las ventajas y desventajas de este proceso, el cual comienza escribiendo de cero una estructura o traduciendo la misma del lenguaje Haskell a *Coq*, e incluso hasta las demostraciones que se realizarán con el asistente de pruebas.

1.3. Traducción de Haskell a *Coq*

El enfoque que hemos decidido darle a este trabajo, consiste en considerar una implementación de la estructura descrita en la sección pasada y solamente realizar la verificación formal de la misma, es por esta razón que como mencionamos en la sección 1.1 de este trabajo, existen programas llamados asistentes de pruebas que nos ayudan a realizar la verificación formal de otros programas, en particular en este trabajo se decidió usar *Coq*. Este asistente nos guía a través de la prueba recordando cuales son las *metas*[4] que debemos demostrar,

[4]Una meta es..

nos ofrece *tácticas*[5] para poder demostrar dichas metas entre otras cosas, sin embargo, esta también nos presenta nuevos desafíos.

Al comenzar a utilizar el asistente nos encontramos con las primeras problemáticas, estas consisten en decidir si vamos a escribir el programa directamente en el lenguaje de *Coq* o si lo que buscamos es traducir un programa ya existente al lenguaje del asistente de pruebas; como lo que se busca es poder verificar un programa antes escrito, es decir, para el caso de los árboles roji-negros queremos traducir este código de Haskell a *Coq*.

En el articulo 'Total Haskell is Reasonable Coq' [Spector-Zabusky et al., 2017] se describen las principales ventajas y desventajas de traducir de Haskell a *Coq*, los cuales expondremos a continuación.

1.3.1. Ventajas

- Haskell es un excelente lenguaje para escribir programas funcionales puros.
- La gran comunidad de programadores que usan y mantienen Haskell.
- El compilador GHC de Haskell, el cual es muy usado e incluso a nivel industrial.
- El ambiente de *Coq* es "amigable" para desarrollar demostraciones formales.
- *Coq* permite razonar acerca de programas funcionales totales.

1.3.2. Desventajas

- Haskell utiliza el razonamiento ecuacional, por lo que en general no se usa un lenguaje formal para realizar las demostraciones.
- Los programadores de Haskell razonan acerca de su código informalmente, si se llegan a realizar pruebas de este, generalmente esta hecho a mano "en papel", lo cual es tedioso y susceptible a errores.
- *Coq* no tiene la extensa biblioteca de funciones ni la misma cantidad de programadores que lo usen y mantengan como lo tiene Haskell.
- El hecho de que los programadores de Haskell sólo razonen acerca de su código informalmente puede que resulte en que se generen funciones parciales, es decir, que no se cubran todas la combinaciones de parámetros posibles para una función.
- La traducción de Haskell a *Coq* sólo es posible si todas las funciones a traducir son totales.

[5] Una táctica es...

Este artículo propone el uso de una herramienta llamada *hs-to-coq*, la cual actualmente se encuentra en etapa de desarrollo y está siendo usada para traducir código de Haskell a *Coq*. Por las razones expuestas al comienzo de esta sección es que decidimos usar esta herramienta de traducción y enfocarnos únicamente en la verificación de los árboles roji-negros

1.4. Sobre este trabajo

El contenido y demostraciones que se describen en este trabajo se encuentran almacenados en: `https://github.com/DavidFHCh/Tesis-FTW`. Aquí presentamos definiciones, lemas y clases sin incluir las demostraciones en *Coq*, en otras palabras, los scripts de prueba. En su lugar se describen de manera informal las demostraciones para poder entender en alto nivel la estructura de la verificación formal realizada.

En este trabajo se optó por usar el traductor *hs-to-coq*, ya que la traducción manual resultaria ser muy tediosa y esta es susceptible a errores, también se desviaría el enfoque de este trabajo, el cual es la verificación de la estructura, no la traducción de la misma. La herramienta fue utilizada para obtener las traducciones de las bibliotecas de Haskell; estas fueron usadas para poder verificar la implementación, comenzando con el trabajo de [López Campos, 2015] y adecuándolo a la implementación de [Appel and Letouzey, 2011] que se uso en el trabajo.

En los siguientes capítulos se describe el procedimiento usado para la verificación de la estructura de datos, la cual esta enfocada a las invariantes de los cuatro puntos de la definición 1.2.1 dejando de lado la verificación de ser un árbol de búsqueda[6].

[6]El lector interesado puede revisar [Appel, 2018].

Capítulo 2

Implementación de arboles roji-negros en *Coq*

2.1. Traducción de implementaciones

Se tomaron un par de implementaciones funcionales de árboles roji-negros para verificarlas: la primera contiene las implementaciones de [López Campos, 2015] en Haskell y que fueron utilizadas como entrada para la herramienta *hs-to-coq*, es decir, una traducción directa. La segunda implementación y la que se usó para este trabajo, fue obtener de [Appel and Letouzey, 2011] la implementación de los árboles roji-negros que se usan en la biblioteca estándar de *Coq*, los cuales son una versión de los árboles roji-negros de Okasaki[Okasaki, 1998]. En el segundo caso, se usaron las bibliotecas traducidas de Haskell a *Coq*, las cuales contienen los tipos y operaciones sobre ellos de Haskell. Esta traducción se obtuvo con la ayuda del traductor *hs-to-coq* y sustituyen a los tipos y operaciones de *Coq*. A continuación profundizaremos de estos dos casos.

2.1.1. Traducción directa de implementaciones de Haskell a *Coq*

De la tesis [López Campos, 2015] se obtuvieron diversas implementaciones de árboles roji-negros; la operación de borrado es compleja (eliminación de un nodo interno) ya que se deben cumplir las invariantes del árbol resultante, es por ello que dicha operación es significativamente mas compleja que su contraparte, es decir, la operación de inserción. Las diferentes implementaciones de la tesis [López Campos, 2015], las cuales buscan una solución mas eficiente, son: la implementación de Okasaki, siendo esta la más simple, los constructores inteligentes (implementación anterior con optimizaciones) y los tipos anidados (una implementación totalmente diferente a las anteriores y mas elegante).

Por la compleja naturaleza de estas implementaciones, la traducción manual del código de Haskell resultó ser muy problemática, esto porque las implemen-

taciones en Haskell se aprovechan del hecho de que en este lenguaje se pueden declarar funciones parciales, lo cual representa un reto al momento de intentar traducir a *Coq*, ya que este lenguaje únicamente acepta funciones totales. Se buscaron soluciones para 'totalizar' estas funciones, sin embargo, solo traerían problemas al intentar realizar las demostraciones, ya que al totalizar se generarían casos inalcanzables en la ejecución, pero tendrían que ser demostrados para poder completar la verificación.

A pasear de ello, se intentó totalizar las funciones de Haskell y así poder usar la herramienta *hs-to-coq* y de esta manera facilitar la traducción, pero por las mismas razones antes descritas, la herramienta caía en alguna de estas dos situaciones:

- El tiempo de ejecución de la herramienta era muy alto y eventualmente los recursos de la maquina virtual, donde esta herramienta se ejecutó, se quedaba sin recursos (en especial memoria). Esto probablemente se deba a la falta de totalidad en alguna función.
- La herramienta generaba código en *Coq* pero con elementos de Haskell cuyas bibliotecas todavía no habían sido traducidas del todo. Esto porque las implementaciones en Haskell podían llegar a ser muy complejas y utilizar módulos de GHC, a los cuales todavía no se les había traducido con la herramienta.

Por estas razones se buscó otro acercamiento para poder verificar esta estructura, entonces, sabemos que el equipo de desarrollo de la herramienta *hs-to-coq* ha traducido exitosamente una fracción de las bibliotecas de Haskell a *Coq*[1], por esta razón, se optó por el uso de la implementación de árboles roji-negros de las bibliotecas de *Coq*, [Appel and Letouzey, 2011], pero usando los tipos y operaciones obtenidos de las traducciones con la herramienta.

2.2. Inserción de elementos en un árbol roji-negro

La inserción de elementos a un árbol roji-negro es la operación mas sencilla de las dos que se verificarán en este trabajo. La idea principal detrás de este algoritmo es que únicamente se agreguen hojas al árbol binario y se efectúen "giros" o funciones de balanceo para mantener las invariantes de la estructura (ver figura 2.1 y 2.2).

2.2.1. Operaciones de Balanceo

Los giros antes mencionados están definidos en las operaciones de balanceo, las cuales son dos, una para los subárboles izquierdos y otra para los derechos. Estas funciones (ver figura 2.3) se encargan de solucionar los casos en los que inmediatamente después de agregar una hoja alguna de las invariantes sean

[1]El nombre de esta es *GHC.Base*, nos referiremos a las funciones de la biblioteca con ese mismo prefijo.

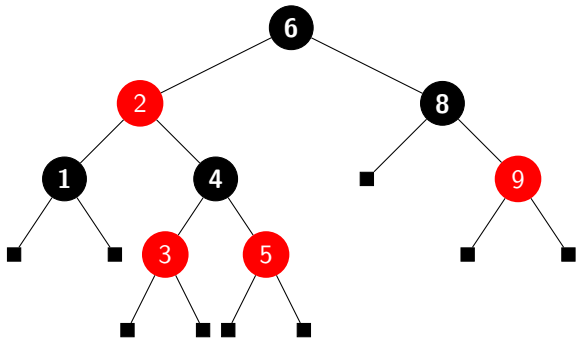

Figura 2.1: Árbol Roji-negro antes de insertar nodo 7.

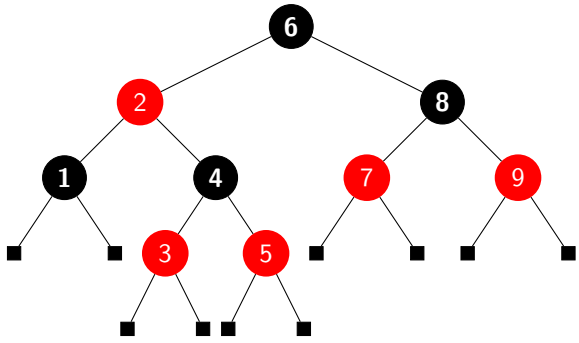

Figura 2.2: Árbol Roji-negro después de insertar nodo 7.

violadas, por ejemplo, dos nodos rojos que resultan contiguos en algún lugar de la estructura del árbol.

```
Definition lbal {a} `{GHC.Base.Ord a} (l:RB a) (k:a) (r:RB a) :=
 match l with
 | T R (T R a x b) y c => T R (T B a x b) y (T B c k r)
 | T R a x (T R b y c) => T R (T B a x b) y (T B c k r)
 | _ => T B l k r
 end.

 Definition rbal {a} `{GHC.Base.Ord a} (l:RB a) (k:a) (r:RB a) :=
 match r with
 | T R (T R b y c) z d => T R (T B l k b) y (T B c z d)
 | T R b y (T R c z d) => T R (T B l k b) y (T B c z d)
 | _ => T B l k r
 end.
```

Figura 2.3: Funciones de Balanceo.

El balanceo elimina el doble nodo rojo al crear únicamente un nodo rojo con dos hijos negros, de igual manera esto nos asegura que el árbol crece de forma controlada en número de nodos negros, es decir, mantiene la invariante de la altura negra, lo cual se debe a que en ningún momento se están agregando dos nodos negros contiguos[2]; cabe mencionar que esta es la única operación en donde se agregan nodos negros, con la excepción de *makeBlack*, la cual describiremos más adelante.

En puntos posteriores se explicarán los casos de uso de esta función, se desarrollará el porqué los únicos casos a los que se les da un trato especial es a los de nodos rojos contiguos y en el resto sólo se regresa un árbol con raíz negra sin hacer mayor acomodo.

2.2.2. Función de inserción

Esta función es donde se presenta por primera vez el uso de las bibliotecas traducidas de Haskell, podemos apreciar como los tipos [3] de los elementos que se están agregando al árbol son tipos ordenados de la biblioteca *Base* del compilador de GHC y por esa misma razón estamos usando las operaciones (comparaciones) de esa biblioteca.

```
Fixpoint ins {a} `{GHC.Base.Ord a} (x:a) (s:RB a) :=
 match s with
 | E => T R E x E
 | T c l y r =>
    if x GHC.Base.< y : bool then
      match c with
       | R => T R (ins x l) y r
       | B => lbal (ins x l) y r
      end
    else
    if x GHC.Base.> y : bool then
      match c with
       | R => T R l y (ins x r)
       | B => rbal l y (ins x r)
      end
    else s
 end.
```

Figura 2.4: Función ins.

Analizando más detenidamente la función (figura 2.4) se puede observar que las operaciones de balanceo solo se efectúan cuando el nodo por el que se esta pasando es negro, esto sucede por la razón de que los nodos de este color son los

[2]Nodos padre e hijo negros después de balancear.
[3]El tipo que se usa en los árboles roji-negros es representado con la letra ***a***.

que se toman en cuenta para decidir si un árbol cumple con el balanceo adecuado. Al aplicar el balanceo en estos nodos, podemos garantizar que no quedarán con nodos negros extras alguno de los hijos de este nodo, es decir, que ninguno de los caminos de la raíz a las hojas tenga mas nodos negros que los demás. Esto se puede apreciar si recordamos lo que se menciono en las definiciones de las operaciones de balanceo, tomemos *rbal* (figura 2.3)[4], tenemos dos casos:

- Sean $\boldsymbol{x}$, $\boldsymbol{y}$ y $\boldsymbol{z}$ nodos del árbol y sea $\boldsymbol{t}$ un subárbol, $\boldsymbol{x}$ es el nodo al que se le aplica la operación de balanceo y este es de color negro, $\boldsymbol{t}$ es el subárbol izquierdo, $\boldsymbol{y}$ es el nodo derecho de $\boldsymbol{x}$ y $\boldsymbol{z}$ es hijo de $\boldsymbol{y}$ (Es irrelevante si es derecho o izquierdo, el resultado es el mismo.). Suponiendo que $\boldsymbol{y}$ y $\boldsymbol{z}$ son rojos[5], se cae en cualquiera de los dos casos de *rbal* que no sean el caso general. En este momento es donde se efectúa el *balanceo* del árbol y resulta lo siguiente: $\boldsymbol{x}$ se vuelve el hijo izquierdo de $\boldsymbol{y}$ y $\boldsymbol{z}$ se pinta de negro [6], todas las demás estructuras del árbol permanecen igual.

  En el momento en que $\boldsymbol{x}$ se convierte en hijo izquierdo de $\boldsymbol{y}$ el árbol se desbalancea, es por esto que se pinta de negro a $\boldsymbol{z}$, así los dos nodos negros son hijos de $\boldsymbol{y}$ y la invariante se conserva.

- En cualquier otro caso el árbol no sufre modificación alguna.

Este balanceo es necesario en esta función, ya que todos los elementos nuevos que se agregan al árbol son hojas rojas, esto puede traer consigo violaciones a los invariantes, en especial al de que existan dos nodos rojos contiguos y esta opearción ayuda a mitigar este problema.

A pesar de que las operaciones de balanceo cuidan la mayoría invariantes en el cuerpo del árbol, la función *ins* no necesariamente cumple con uno de los invariantes, específicamente en el que la raíz del árbol es negra, es por ello que se introducen las definiciones de la figura 2.5.

```
Definition makeBlack {a} `{GHC.Base.Ord a} (t:RB a) :=
 match t with
 | E => E
 | T _ a x b => T B a x b
 end.

Definition insert {a} `{GHC.Base.Ord a} (x:a) (s:RB a) :=
                                          makeBlack (ins x s).
```

Figura 2.5: Definiciones para pintar raíz de negro.

La definición *makeBlack* únicamente colorea un nodo de color negro y la definición *insert* es una envoltura de *ins*, con la cual nos aseguramos de que

[4] Con *lbal* la idea es análoga
[5] se viola una invariante, dos nodos rojos contiguos
[6] El hijo se vuelve padre y el padre se vuelve hijo.

la raíz de los árboles siempre sea de color negro, esto se logra con ayuda de *makeBlack*.

Estas funciones y definiciones son suficientes para poder construir árboles roji-negros que respeten las invariantes que planteamos en la definición 1.2.1.

2.3. Eliminación de elementos en un árbol roji-negro

Como se mencionó en la sección anterior, la operación de eliminación es significativamente más compleja que su contra parte, esto se debe al hecho de que puede ser eliminado cualquier nodo en un árbol roji-negro, mientras que en la inserción sólo se agrega el elemento como una hoja de color rojo, es decir, la altura únicamente se modifica en la inserción cuando se aplica el balanceo.

La acción de eliminar un nodo de cualquier parte de un árbol roji-negro presenta una problemática muy grande para mantener las invariantes de los árboles roji-negros, esto se suscita al eliminar un nodo interno del árbol derivando en dos subárboles que tienen que ser concatenados de alguna forma.

2.3.1. Función de eliminación

Para poder comprender la lógica de las funciones que conforman a la operación de eliminación es necesario comenzar por la función que retira el nodo del árbol (ver la figura 2.6). La idea central de esta operación es bastante simple: como los árboles roji-negros son árboles de búsqueda, lo primero que hacemos es buscar el nodo a eliminar, si se encuentra se elimina y se concatenan los subárboles restantes de esta operación (ver figuras 2.7, 2.8 y 2.9). A continuación se describen más a fondo los casos de la misma:

- Si se recibe un árbol vacío como argumento de la función, se regresa este mismo; pues eliminar un elemento del árbol vacío termina siendo vacío. También este caso sirve para cuando un elemento no es encontrado en el árbol, es el caso base de la recursión de búsqueda del nodo a eliminar.

- En otro caso, se realiza recursivamente la búsqueda del elemento a eliminar. Si el nodo actual no contiene el elemento que buscamos, se compara si es menor o mayor para seguir buscando en el árbol izquierdo o derecho respectivamente, describimos los casos:

    - Si el nodo es menor y el nodo del árbol izquierdo es negro, entonces aplicamos la función $lbalS$ [7] al árbol resultante de seguir buscando el nodo a eliminar por el subárbol izquierdo. En otro caso solo seguimos buscando por el subárbol izquierdo.

[7] Función de balanceo extendida para subárboles izquierdos.

- Si el nodo es mayor y el nodo del árbol derecho es negro, entonces aplicamos la función *rbalS* [8] al árbol resultante de seguir buscando el nodo a eliminar por el subárbol derecho. En otro caso solo seguimos buscando por el subárbol derecho.
- Si el elemento en el que estamos no es ni mayor ni menor al que buscamos, en ese caso eliminamos el elemento y concatenamos los subárboles restantes usando la función *append* [9].

```
Fixpoint del {a} `{GHC.Base.Ord a} (x:a) (t:RB a) :=
 match t with
 | E => E
 | T _ a y b =>
    if x GHC.Base.< y : bool then
      match a with
       | T B _ _ _ => lbalS (del x a) y b
       | _ => T R (del x a) y b
      end
    else
    if x GHC.Base.> y : bool then
      match b with
       | T B _ _ _ => rbalS a y (del x b)
       | _ => T R a y (del x b)
      end
    else append a b
 end.

Definition remove x t := makeBlack (del x t).
```

Figura 2.6: Función de eliminación

Podemos ver que las funciones de balanceo *lbalS* y *rbalS* se aplican cuando el nodo en el que estamos parados, llamémoslo $\boldsymbol{n}$, es negro; esto evita que después de eliminar un nodo y aplicar la función *append* se acabe con dos nodos rojos seguidos, es decir, que el hijo y alguno de los nietos del nodo $\boldsymbol{n}$ sean rojos.

2.3.2. Función de concatenación (*append*)

La función de concatenación (figura 2.10) es usada cuando se ha encontrado el elemento a eliminar en un árbol roji-negro, resultando en dos árboles que tienen que ser concatenados, los cuales deben de respetar los invariantes de los árboles roji-negros. Esta función recibe como parámetros los dos árboles[10] que estamos buscando unir. Esta operación se describe con mayor detalle en seguida.

[8] Función de balanceo extendida para subárboles derechos.
[9] Función donde se juntan lo arboles restantes de esta operación
[10] Estos arboles pueden no cumplir con todas las invariantes de los árboles roji-negros.

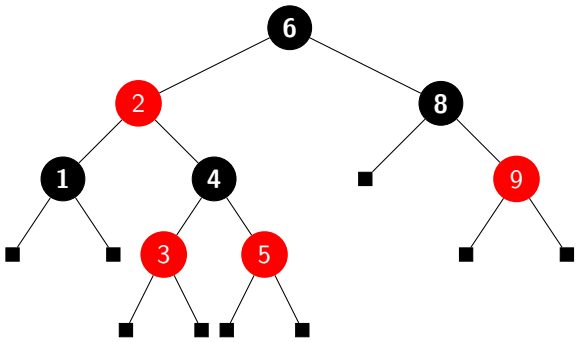

Figura 2.7: Árbol Roji-negroantes de eliminar nodo 6.

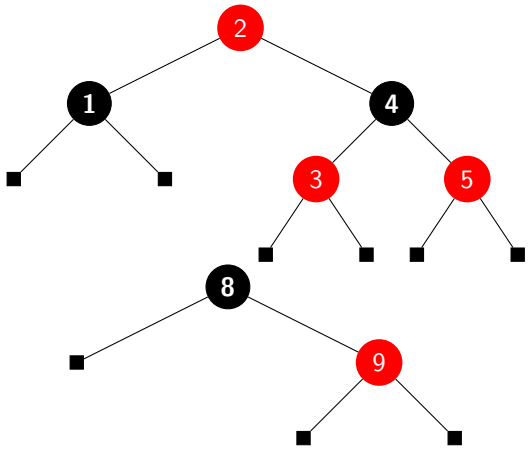

Figura 2.8: Árbol Roji-negro roto, después de eliminar nodo 6.

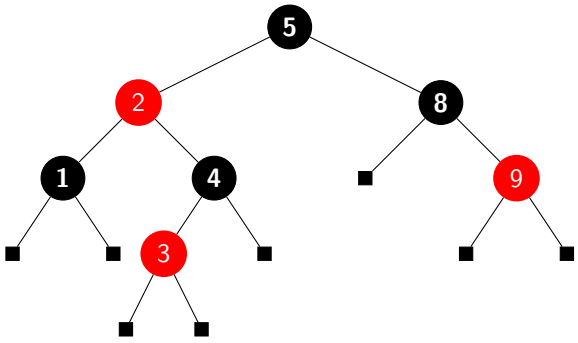

Figura 2.9: Árbol Roji-negro después de aplicar función append.

Sean ***a*** y ***b*** los dos subárboles a los que se les aplicará la función append, es decir, *append a b*, tenemos los siguientes casos que están definidos dentro de la

```
Fixpoint append {a} `{GHC.Base.Ord a} (l:RB a) : RB a -> RB a :=
 match l with
 | E => fun r => r
 | T lc ll lx lr =>
   fix append_l (r:RB a) : RB a :=
   match r with
   | E => l
   | T rc rl rx rr =>
     match lc, rc with
     | R, R =>
       let lrl := append lr rl in
       match lrl with
       | T R lr' x rl' => T R (T R ll lx lr') x (T R rl' rx rr)
       | _ => T R ll lx (T R lrl rx rr)
       end
     | B, B =>
       let lrl := append lr rl in
       match lrl with
       | T R lr' x rl' => T R (T B ll lx lr') x (T B rl' rx rr)
       | _ => lbalS ll lx (T B lrl rx rr)
       end
     | B, R => T R (append_l rl) rx rr
     | R, B => T R ll lx (append lr r)
     end
   end
 end.
```

Figura 2.10: Función de concatenación, append

función auxiliar local (*append_l*):

- Si ***a*** es el árbol vacío, entonces se regresa ***b***.
- Si ***b*** es el árbol vacío, entonces regresamos ***a***.
- Si ***a*** y ***b*** son árboles con raíces rojas, entonces se aplica *append* al subárbol derecho de ***a***, sea este ***ar***, junto con el subárbol izquierdo de ***b***, sea ***bl***, es decir, *append ar bl*. Tenemos subcasos:
  - Si el resultado de esta operación es un árbol con raíz roja, sea ***arbl***, los árboles ***a*** y ***b*** se pintan de rojo y se concatenan con la raíz de ***arbl***, igual de color rojo; ***ar*** se reemplaza por el subárbol izquierdo de ***arbl*** y ***bl*** se reemplaza por el subárbol derecho de ***arbl***.
  - En otro caso, si el árbol resultante de *append ar bl* no es rojo, tomamos ***a*** y ***b***, los pintamos de rojo, el subárbol derecho de ***a*** se

reemplaza por $\boldsymbol{b}$ y el subárbol izquierdo de $\boldsymbol{b}$ se reemplaza por el resultado de *append ar bl*.

- Si $\boldsymbol{a}$ y $\boldsymbol{b}$ son arboles con raíces negras, entonces se aplica *append* al subárbol derecho de $\boldsymbol{a}$, sea $\boldsymbol{ar}$, con el subárbol izquierdo de $\boldsymbol{b}$, sea $\boldsymbol{bl}$, es decir, *append ar bl*. Tenemos casos:
  - Si el resultado de esta operación es un árbol con raíz roja, sea $\boldsymbol{arbl}$, los árboles $\boldsymbol{a}$ y $\boldsymbol{b}$ se pintan de negro y se concatenan con la raíz de $\boldsymbol{arbl}$, esta de color rojo; $\boldsymbol{ar}$ se reemplaza por el subárbol izquierdo de $\boldsymbol{arbl}$ y $\boldsymbol{bl}$ se reemplaza por el subárbol derecho de $\boldsymbol{arbl}$.
  - En otro caso, si el árbol resultante de *append ar bl* no es rojo, tomamos $\boldsymbol{a}$ y $\boldsymbol{b}$, el subárbol derecho de $\boldsymbol{a}$ se reemplaza por $\boldsymbol{b}$ y el subárbol izquierdo de $\boldsymbol{b}$ se reemplaza por el resultado de *append ar bl* y a este resultado le aplicamos una función de balanceo, $lbalS$.
- Si $\boldsymbol{a}$ es un árbol de color negro y $\boldsymbol{b}$ de color rojo, entonces se toma $\boldsymbol{b}$ rojo pero en lugar de su subárbol izquierdo, se aplica una llamada recursiva con $\boldsymbol{bl}$ (subárbol izquierdo de $\boldsymbol{b}$) a la función embebida en *append*, llamada *append_l*, es decir: *append_l bl*, esta llamada también carga al árbol $\boldsymbol{a}$ gracias al currying[HaskellWiki, 2020].
- Si $\boldsymbol{a}$ es un árbol de color rojo y $\boldsymbol{b}$ de color negro, entonces se toma $\boldsymbol{a}$, se pinta de rojo pero en lugar de su subárbol derecho, sea $\boldsymbol{ar}$, se hace una llamada recursiva con $append(ar, b)$.

Debemos mencionar que el árbol resultante de aplicar esta función no necesariamente cumple con todos las invariantes de un árbol roji-negro, estas invariantes se logran conservar ya que en la función *del* se realizan llamadas a las funciones extendidas de balanceo, las cuales desarrollaremos en la siguiente sección.

2.3.3. Extensión de funciones de balanceo

En la sección 2.2 de este trabajo se trató la inserción de elementos a un árbol roji-negro, en donde se describen un par de funciones llamadas 'de balanceo', tratadas en las subsección 2.2.1; estas funciones a su vez toman los nombres *rbal* y *lbal* (figura 2.3). Estas operaciones resultan insuficientes para balancear un árbol al momento de eliminar un nodo y concatenar los dos árboles restantes con la función *append*, es por esto que se implementan las extensiones de estas funciones, llamadas $lbalS$ y $rbalS$ (figuras 2.11 y 2.12 respectivamente) las cuales a su vez llaman a las funciones $rbal'$[11] (figura 2.13) y *lbal*. Estas extensiones agregan mas casos de manejo de subárboles negros, esto sucede porque existen casos en los que se puede llegar a eliminar un nodo negro intermedio del árbol, inclusive la misma raíz y se debe poder asegurar que las invariantes no se violen después de concatenar los subárboles restantes.

[11] La función $rbal'$ es una variación de la función *rbal*, solo cambia el orden de la caza de patrones.

```
Definition lbalS {a} `{GHC.Base.Ord a} (l:RB a) (k:a) (r:RB a) :=
 match l with
 | T R a x b => T R (T B a x b) k r
 | _ =>
   match r with
   | T B a y b => rbal' l k (T R a y b)
   | T R (T B a y b) z c => T R (T B l k a) y (rbal' b z (makeRed c))
   | _ => T R l k r
   end
 end.
```

Figura 2.11: Función de balanceo de lado izquierdo extendida.

Las funciones *rbalS* y *lbalS* son usadas en la función *del* (figura 2.6) cuando la llamada recursiva busca el nodo a eliminar en un subárbol con raíz negra, al aplicar la función de balanceo en estos nodos podemos asegurar que los dos subárboles de este nodo no se van a desequilibrar, es decir, que un subárbol tenga mayor altura negra que el otro. Estas funciones básicamente hacen un reacomodo de los nodos de los subárboles para después llamar a las funciones *rbal′* y *lbal*, es por eso que decimos que son extensiones de estas.

```
Definition rbalS {a} `{GHC.Base.Ord a} (l:RB a) (k:a) (r:RB a) :=
 match r with
 | T R b y c => T R l k (T B b y c)
 | _ =>
   match l with
   | T B a x b => lbal (T R a x b) k r
   | T R a x (T B b y c) => T R (lbal (makeRed a) x b) y (T B c k r)
   | _ => T R l k r
   end
 end.
```

Figura 2.12: Funciones de balanceo de lado derecho extendida.

Existe otra función donde se utiliza una de estas operaciones de balanceo, específicamente *lbalS*, esta función es *append*. Esto sucede en el caso en que los árboles que son argumentos de *append* sean negros, es decir, la misma razón por la que se aplican las funciones de balanceo en *del* sobre los nodos de color negro: para que sus subárboles no se desbalanceen.

Las definiciones y funciones presentadas son suficientes para poder eliminar cualquier nodo de un árbol roji-negro y que el resultado no viole los invariantes de estos. En esta etapa del trabajo nos agradaría llegar a esta conclusión, sin embargo, esta sentencia tiene que ser demostrada, es decir, tenemos que probar que nuestros árboles roji-negros cumplen con la definición de los mismos. Esto es lo que veremos en el siguiente capítulo.

```
Definition rbal' {a} `{GHC.Base.Ord a} (l:RB a) (k:a) (r:RB a) :=
 match r with
 | T R b y (T R c z d) => T R (T B l k b) y (T B c z d)
 | T R (T R b y c) z d => T R (T B l k b) y (T B c z d)
 | _ => T B l k r
 end.
```

Figura 2.13: Función de balanceo de lado derecho alternativa.

Capítulo 3

Verificación de árboles roji-negros

En el primer capítulo de este trabajo se mencionó que los árboles roji-negros son una estructura de datos que mejora el tiempo de acceso, de inserción y eliminación de elementos con respecto a otras estructuras de datos como: las listas simples, las listas doblemente ligadas y árboles de búsqueda. En el segundo capítulo se muestran las implementaciones funcionales de los algoritmos de esta estructura de datos y podemos notar como estas implementaciones no son triviales, es decir, son rebuscadas por los tipos usados en un lenguaje que utiliza el estilo de programación funcional como lo hace *Coq*, e incluso son diferentes y alejadas de una implementación en un lenguaje con un paradigma imperativo como *Java* o *C*. Por esta razón es que nos preocupa que las implementaciones que realicemos sean correctas, en otras palabras, se desea verificar que las implementaciones de las operaciones descritas en el capitulo anterior respeten en todos los casos los invariantes de los árboles roji-negros.

3.1. Pruebas unitarias

Las *pruebas unitarias* [Osherove, 2014] son bloques de código, funciones o métodos, que invocan a otros bloques para poder verificar ciertas suposiciones sobre el programa a probar. Estas pruebas en principio deben de ser fáciles de escribir, entender, extender, que se ejecuten en poco tiempo y sobre todo que sean fidedignas. De nada nos servirían pruebas unitarias que estén mal escritas o que estas mismas sean demasiado complejas y puedan contener errores.

Este tipo de pruebas son usadas para verificar que cada componente de un programa funcione de manera esperada, en el caso de los árboles roji-negros este tipo de pruebas nos ayudan a verificar los invariables de un determinado árbol. La figura 3.1 es una prueba unitaria escrita en el lenguaje de programación Java, la cual verifica la altura negra de un árbol roji-negro [Pelaez, 2019].

```java
/* Valida que los caminos del vértice a sus hojas tengan todos
   el mismo número de vértices negros. */
private static <T extends Comparable<T>> int
validaCaminos(ArbolRojinegro<T> arbol,
              VerticeArbolBinario<T> v) {
    int ni = -1, nd = -1;
    if (v.hayIzquierdo()) {
        VerticeArbolBinario<T> i = v.izquierdo();
        ni = validaCaminos(arbol, i);
    } else {
        ni = 1;
    }
    if (v.hayDerecho()) {
        VerticeArbolBinario<T> d = v.derecho();
        nd = validaCaminos(arbol, d);
    } else {
        nd = 1;
    }
    Assert.assertTrue(ni == nd);
    switch (arbol.getColor(v)) {
    case NEGRO:
        return 1 + ni;
    case ROJO:
        return ni;
    default:
        Assert.fail();
    }
    // Inalcanzable.
    return -1;
}
```

Figura 3.1: Prueba unitaria escrita en Java.[Pelaez, 2019]

Sin embargo, el hecho de que las pruebas unitarias puedan verificar los invariantes de un árbol dado, no nos asegura que todos los árboles creados por nuestras operaciones de inserción y eliminación los respeten. La única manera de que esta prueba podría verificar esto sería realizando todas las combinaciones de operaciones y entradas posibles y aplicar la prueba a todos los resultados de estas entradas. Esto es una prueba exhaustiva y en este caso [1] las posibilidades son infinitas, es decir, no existe modo de realizar todas estas pruebas en un tiempo razonable, más aún, no existe memoria suficiente para almacenar los infinitos valores que se necesitarían para realizar dichas pruebas.

Finalmente, como las pruebas unitarias no nos permiten cubrir todos los

[1] De hecho en la mayoría.

casos necesarios para poder probar en su totalidad un programa, es que se requiere utilizar algún otro tipo de herramienta para realizar una verificación, en este trabajo usaremos *Coq*.

3.2. Verificación Formal en *Coq*

Es claro que las pruebas unitarias no nos son suficientes para poder verificar formalmente un programa, es por esto que se requieren realizar demostraciones matemáticas para poder obtener los resultados que buscamos, pero de igual manera no queremos escribir a mano estas demostraciones ya que al igual que las pruebas unitarias estas son susceptibles al error humano, por esta razón es que es mejor que un sistema como *Coq* guíe la demostración para verificar cada paso, ya que a su vez esta herramienta esta verificada para asegurar que no se incluyan errores y también para garantizar la consistencia de las pruebas[Geuvers, 2009].

La verificación formal de un programa usando *Coq* esta compuesto de las siguientes etapas:

- Capturar los invariantes o propiedades de un programa usando definiciones inductivas en *Coq*, de esta manera podemos saber si las operaciones que se implementaron las respetan. Las operaciones pueden o no estar programadas en *Coq*.
- Enunciar lemas, corolarios y teoremas que describan los comportamientos de las operaciones que queremos verificar y escribirlos en *Coq*, usando las definiciones inductivas descritas en el punto anterior.
- Por último, demostrar todos los enunciados del punto anterior utilizando las tácticas que el asistente de pruebas nos provee.

3.2.1. Capturando invariantes de los Árboles Roji-negros

Una de las etapas más importantes al realizar la verificación formal en *Coq* de cualquier estructura de datos, inclusive de cualquier programa, es capturar sus invariantes de manera correcta, es decir, poder escribir una o varias definiciones inductivas que describan a la estructura de datos y sus invariantes. Después, con estas mismas es que se enuncian los lemas, clases y posteriormente instancias de las clases.

A continuación se describen dos conjuntos de definiciones inductivas, muy similares entre ellas[2], los cuales nos ayudaran a verificar formalmente los árboles roji-negros. La primera es un primer intento que es insuficiente ya que los tipos inductivos y los principios de demostración no son los óptimos. El segundo intento es un conjunto de definiciones inductivas que tienen mas detalle para describir los invariantes. Estas definiciones están relacionadas con las propiedades de las operaciones de inserción y eliminación.

[2]Obtenidas de los trabajos [López Campos, 2015] y [Appel and Letouzey, 2011] respectivamente.

Es importante mencionar que estas definiciones aprovechan la expresividad de la lógica o de los tipos de *Coq* ya que se están usando a su vez otros tipos inductivos para describir las invariantes.

Primer Conjunto de Definiciones Inductivas

Los dos conjuntos de definiciones inductivas comparten la misma idea: una definición que describe estrictamente lo que es un árbol roji-negro y otra definición más laxa de la misma. Utilizar una definición mas restrictiva que otra es una implementación, hasta cierto punto, de la técnica *divide y vencerás*[3], la cual es una manera de resolver problemas mas complejos dividiéndolo en partes mas pequeñas. En este caso la división se realiza en las invariantes.

```
Inductive isRB : Tree -> color -> nat -> Prop :=
 | IsRB_leaf: forall c, isRB E c 0
 | IsRB_r: forall tl k kv tr n,
          isRB tl Red n ->
          isRB tr Red n ->
          isRB (T Red tl k kv tr) Black n
 | IsRB_b: forall c tl k kv tr n,
          isRB tl Black n ->
          isRB tr Black n ->
          isRB (T Black tl k kv tr) c (S n).
```

Figura 3.2: Función inductiva isRB.

La primera definición llamada $isRB$ (figura 3.2) tiene tres casos, los cuales describiremos a continuación:

- **IsRB_Leaf**: el árbol vacío con altura negra 0 es roji-negro. En este caso se tiene un sólo nodo, es decir, una hoja[4].
- **IsRB_r**: Para cualesquiera árboles $\boldsymbol{tl}$ y $\boldsymbol{tr}$ que cumplan con la definición $isRB$ con color rojo y altura $\boldsymbol{n}$, se cumple que un árbol de color rojo con $\boldsymbol{tl}$ y $\boldsymbol{tr}$ como subarboles, sea $\boldsymbol{t}$, cumple con $isRB$ con color negro y altura $\boldsymbol{n}$. El color se refiere al color del padre, en este caso, en la llamada de $isRB$ a $\boldsymbol{tl}$ y $\boldsymbol{tr}$ se le pasa el color rojo porque $\boldsymbol{t}$ es rojo. La altura $\boldsymbol{n}$ se refiere a que hay $\boldsymbol{n}$ nodos negros en cualquier camino del nodo actual a alguna hoja, aqui no crece $\boldsymbol{n}$ porque tanto $\boldsymbol{tl}$ y $\boldsymbol{tr}$ son rojos.
- **IsRB_b**: Para cualesquiera árboles $\boldsymbol{tl}$ y $\boldsymbol{tr}$ que cumplan con la definición $isRB$ con color negro y altura $\boldsymbol{n}$, se cumple que un árbol de color negro con $\boldsymbol{tl}$ y $\boldsymbol{tr}$ como subarboles, sea $\boldsymbol{t}$, cumple con $isRB$ con cualquier color y altura $S(n)$. En este caso, en la llamada de $isRB$ a $\boldsymbol{tl}$ y $\boldsymbol{tr}$ se le pasa el

[3] *Divide and conquer* en ingles.
[4] Recordemos que las hojas son vacias y de color negro.

color negro porque ***t*** es negro y aqui se le suma uno a ***n*** porque tanto ***tl*** y ***tr*** son negros.

Con estos tres casos podemos asegurar que las invariantes se respetan, pero esta función inductiva es demasiado restrictiva y esto dificulta poder demostrar las propiedades de los árboles roji-negros, por esto pasamos a la segunda definición inductiva, $nearRB$, esta permite mas flexibilidad en el árbol, se muestra y describe en la figura 3.3.

```
Inductive nearRB : Tree -> nat -> Prop :=
| nrRB_r: forall tl k kv tr n,
         isRB tl Black n ->
         isRB tr Black n ->
         nearRB (T Red tl k kv tr) n
| nrRB_b: forall tl k kv tr n,
         isRB tl Black n ->
         isRB tr Black n ->
         nearRB (T Black tl k kv tr) (S n).
```

Figura 3.3: Función inductiva nearRB.

Podemos apreciar que solo se tienen dos casos y no se tiene un argumento para un color, sin embargo, a diferencia de $isRB$ esta no se llama recursivamente, en lugar de eso se llama a $isRb$ inmediatamente, además podemos ver que ambas definiciones comparten el contador de nodos negros. Con estas modificaciones se permite una cosa, que en la raíz del árbol puedan haber a lo más dos nodos rojos contiguos.

Intento de Verificación Utilizando las definiciones inductivas descritas en esta sección se realizó un intento fallido de verificacón de la operación de inserción, como se muestra en [Appel, 2018], sin embargo, al estar desarrollando la demostración se encontró un problema, la falta de un conjunto de hipótesis para poder probar una meta. Esto se debe probablemente a una mala elección de estilo de demostración, implementación o de las definiciones inductivas mostradas anteriormente. Se noto que el hecho de que toda la información referente a los invariantes estuviera codificada en las dos funciones inductivas, sin uso de "funciones auxiliares" complica la verificación. Se llego a esta conclusión ya que el caso "sencillo" de la verificación de árboles roji-negros es la inserci ón y con este conjunto de funciones inductivas las demostraciones se volvían muy largas y complicadas de seguir.

Segundo Conjunto de Definiciones Inductivas

Con el conocimiento que se obtuvo del conjunto de definiciones anterior, nos realizamos la siguiente pregunta: ¿cómo capturar las invariantes de los árboles roji-negros, y al mismo tiempo facilitar la verificación de estos?

Utilizamos una definición inductiva, llamada *is_redblack* para poder capturar los invariantes, la cual lleva como parámetros un contador y un árbol. El contador lleva el control de la cantidad de nodos negros, es decir, la altura negra del nodo, mientras que el árbol es aquel que estamos buscando verificar que cumpla con las invariantes de un árbol roji-negro. Se presenta esta definición en la figura 3.4.

```
Inductive is_redblack {a} `{GHC.Base.Ord a} : nat -> RB a -> Prop :=
 | RB_Leaf : is_redblack 0 E
 | RB_R n l k r : notred l -> notred r ->
                  is_redblack n l -> is_redblack n r ->
                  is_redblack n (T R l k r)
 | RB_B n l k r : is_redblack n l -> is_redblack n r ->
                  is_redblack (S n) (T B l k r).
```

Figura 3.4: Función inductiva *is_redblack*.

Podemos notar ciertas similitudes con la definición inductiva de la sección pasada, sin embargo, el principal cambio que presenta esta definición, es el hecho de que se dejan de controlar los colores de los subárboles en los parámetros de la definición y se crea la función *notred*, la cual, como su nombre dice, verifica que el árbol que se este pasando no tenga raíz roja. La definición *is_redblack* tiene tres casos,RB_Leaf, RB_R y RB_B. Desarrollando la idea de cada caso:

- **RB_Leaf**: el árbol vació es roji-negro. Este caso nos dice que el árbol vacío es un árbol roji-negro.
- **RB_R**: un árbol rojo donde se lleven contabilizados $\boldsymbol{n}$ nodos negros, donde sus hijos sean árboles roji-negros y no sean rojos. Este caso nos dice explícitamente que los subárboles del árbol que esta recibiendo la función no pueden ser rojos, esto porque el árbol que se esta analizando tiene raíz roja. Como no se esta analizando algún nodo negro, la altura negra se mantiene en $\boldsymbol{n}$.
- **RB_B**: un árbol negro donde se lleven contabilizados $n+1$ nodos negros, incluido el actual, y sus hijos sean árboles roji-negros. En este último caso se tiene la libertad de que los subárboles sean de cualquier color, pero la altura del árbol de regreso es $S(n)$ porque el nodo que se esta analizando es de color negro, los antecedentes al no tomar en cuenta a su nodo padre tienen altura $\boldsymbol{n}$.

Esta definición captura los invariantes que estamos buscando, sin embargo, no es suficiente para poder probar la corrección de los árboles roji-negros, la definición es demasiado restrictiva y costaría mucho trabajo proceder con las demostraciones solamente con ella. Por esta razón se agregan dos definiciones inductivas auxiliares; *redred_tree* y *nearly_redblack* (figura 3.5).

```
Inductive redred_tree {a}
                  `{GHC.Base.Ord a} (n:nat) : RB a -> Prop :=
 | RR_Rd l k r : is_redblack n l -> is_redblack n r
                                 -> redred_tree n (T R l k r).

Inductive nearly_redblack {a}
                  `{GHC.Base.Ord a} (n:nat)(t:RB a) : Prop :=
 | ARB_RB : is_redblack n t -> nearly_redblack n t
 | ARB_RR : redred_tree n t -> nearly_redblack n t.
```

Figura 3.5: Funciones inductivas *redred_tree* y *nearly_redblack*.

Podemos notar que estas definiciones son versiones menos restrictivas de *is_redblack*. La definición *nearly_redblack* permite que existan dos nodos rojos en la raíz del árbol, aprovechándose de *redred_tree*, pues esta definición es exactamente el caso *RB_R* de *is_redblack* pero sin las restricciones de que los subárboles sean rojos, lo cual nos permite que hayan dos nodos rojos exactamente en la raíz. Entonces un *nearly_redblack* es un árbol roji-negro con la excepción de que la raíz puede ser roja. Esto porque si analizamos nuestras funciones de balanceo, no tenemos manera de pintar la raíz de negro, mas que al utilizar la función $make_b lack$ al final de las operaciones que se realicen.

Estas definiciones, a diferencia del conjunto anterior, utilizan mas acentuadamente la técnica de *divide y vencerás*, es decir, se buscan demostrar definiciones mas laxas[5] para al final utilizar estos resultados para realizar la demostración de la definición mas estricta.

Finalmente, lo que se busca demostrar es que los árboles roji-negros con las operaciones de inserción y eliminación estén dentro de la clase de árboles *redblack* (figura 3.6).

```
Class redblack {a} `{GHC.Base.Ord a} (t:RB a) :=
                            RedBlack : exists d, is_redblack d t.
```

Figura 3.6: Clase de árboles *redblack*.

Lo que estamos describiendo con el enunciado de la figura 3.6 es que dado un árbol roji-negro (*t: RB a*), existe una altura negra *d* que cumple con las invariantes establecidas por la definición *is_redblack*. Esta clase[6] representa un refinamiento en los tipos inductivos para describir las invariantes de las estructuras de datos y es mas expresivo que los tipos de datos usados en la implementación de *Haskell*.

[5] Sin todas las invariantes.

[6] Las clases en este contexto funcional se pueden ver, desde un alto nivel, como equivalencias a las clases del paradigma orientado a objetos.

Segundo Intento de Verificación En contraste con el conjunto de definiciones de la sección pasada, la definición de *nearly_redblack* se reescribe, dejando de codificar las invariantes en las llamadas recursivas de la definición, creando funciones auxiliares para capturar los invariantes de manera mas sencilla, como *redred_tree* y *notred*. Además se crea la clase de árboles roji-negros, lo cual afina mas la definición de los mismos. Tomando en cuenta todas estas modificaciones a las definiciones fue que se eligió este conjunto para verificar formalmente la estructura de datos[7]. Estas definiciones inductivas fueron obtenidas de [Appel and Letouzey, 2011].

3.2.2. Verificación de la operación de inserción

Para poder realizar la verificación de la operación de inserción es necesario escribir enunciados, ya sean lemas, proposiciones, etc. Estos enunciados los escribiremos usando las definiciones inductivas presentadas en la sección pasada, es decir, *is_redblack*, *nearly_redblack* y *redred_tree*.

A continuación mostraremos los lemas *ins_rr_rb*, *ins_arb* y una instancia [Castéran and Sozeau, 2016] de la clase *redblack*, *add_rb*. Estos lemas y la instancia fueron obtenidos de [Appel and Letouzey, 2011], la idea principal de estos enunciados es explicarnos que ciertos árboles de búsqueda que respeten las definiciones mas generales, es decir, *nearly_redblack* y *redred_tree*, también como consecuencia respetarán *is_redblack*.

Primer Lema

```
Lemma ins_rr_rb {a} `{GHC.Base.Ord a} (x:a) (s: RB a) (n : nat) :
is_redblack n s ->
     ifred s (redred_tree n (ins x s)) (is_redblack n (ins x s)).
```

Figura 3.7: Lema *ins_rr_rb*

En este primer lema (figura 3.7) enunciamos lo siguiente: sea ***s*** un árbol roji-negro bajo la definición de *is_redblack*, entonces si ***s*** es un árbol con raíz roja, insertar un elemento ***x*** en ***s*** resulta en un árbol que cumple la defincón de *redred_tree*, en otro caso cumple con la definición de *is_redblack*.

En otras palabras, lo que este enunciado quiere decirnos es que si tenemos un árbol roji-negro e insertamos un elemento a ese árbol el resultado puede tener raíz roja, e incluso puede tener dos nodos rojos, uno en la raíz y otro en cualquiera de los dos, o incluso en los dos nodos siguientes.

La demostración de este lema comienza con una inducción sobre el antecedente del enunciado, lo cual resulta en tres casos:

[7] La elección de este conjunto fue correcta ya que facilito la demostración de las propiedades y probo ser suficiente para verificar la estructura.

```
______________________________________(1/3)
ifred E (redred_tree 0 (ins x E)) (is_redblack 0 (ins x E))
______________________________________(2/3)
ifred (T R l k r) (redred_tree n (ins x (T R l k r)))
                  (is_redblack n (ins x (T R l k r)))
______________________________________(3/3)
ifred (T B l k r) (redred_tree (S n) (ins x (T B l k r)))
                  (is_redblack (S n) (ins x (T B l k r)))
```

La función $ifred$ que se usa en este lema, es una función auxiliar que nos ayuda a decidir si un árbol es rojo o no.

En el primero de estos casos notamos que su solución se da simplificando las funciones y resulta en uno de los casos de $is\_redblack$, especificamente en el caso RB_R, ya que el árbol vacío no es rojo y la simplificación de $(ins(x, E))$ resulta en un árbol rojo con un elemento, esto por definición de ins.

Los dos casos sobrantes estan relacionados con los colores de las raices del árbol, en el segundo el árbol es rojo y en el tercero es negro.

Analicemos el segundo caso; como el árbol es rojo entramos al primer caso de if_red, es decir, al caso donde se aplica la definición $redred\_tree$, lo cual significa que al insertar un elemento al árbol rojo, sin tener conocimiento de como son los subárboles de este, puede resultar en un árbol con uno o dos nodos rojos consecutivos en la raíz del mismo, ya que la operación de balanceo se fija en los nodos hijos y nietos del nodo al que se le aplica la operación, y como los nodos hijos de la raíz no tienen nodo abuelo, el balanceo no se efectúa en los nodos de la raíz, dando lugar a arboles con uno o mas nodos rojos consecutivos en la raíz[8].

El caso sobrante, es decir, el caso del árbol con raíz negra se complica un poco mas que el anterior ya que este es el caso en el que la altura negra del árbol se ve modificada, en otras palabras, puede aumentar en uno. Este caso se verifica con las dos funciones de balanceo, *lbal* y *rbal*:

```
H1_ : is_redblack n l
H1_0 : is_redblack n r
IHis_redblack1 :
    ifred l (redred_tree n (ins x l)) (is_redblack n (ins x l))
IHis_redblack2 :
    ifred r (redred_tree n (ins x r)) (is_redblack n (ins x r))
______________________________________(1/2)
is_redblack (S n) (lbal (ins x l) k r)
______________________________________(2/2)
is_redblack (S n) (rbal l k (ins x r))
```

Estos casos son análogos y los dos se resuelven simplificando las funciones de balanceo, en otros términos, simplificando las expresiones y aplicando las definiciones inductivas[9].

[8] hasta 3, la raíz y sus hijos.

[9] $is\_redblack$, $nearly\_redblack$

Segundo Lema

```
Lemma ins_arb {a} `{GHC.Base.Ord a} (x:a) (s:RB a) (n:nat) :
is_redblack n s -> nearly_redblack n (ins x s).
```

Figura 3.8: Lema *ins_arb*

Este segundo lema (figura 3.8) enuncia lo que en el capítulo anterior se menciono acerca de la función *ins*: la función *ins* no garantiza que el árbol resultante sea un árbol roji-negro, ya que es posible que se termine la ejecución de la función con un nodo rojo como raíz. La demostración comienza introduciendo los antecedentes a las hipótesis y aplicando el lema anterior, *ins_rr_rb*, a una de las hipótesis:

```
H1 : ifred s (redred_tree n (ins x s)) (is_redblack n (ins x s))
______________________________________(1/1)
nearly_redblack n (ins x s)
```

Aplicamos el lema *ins_rr_rb* a la hipótesis porque este nos genera dos casos, uno con la función *redred_tree* y otro con *is_redblack*. Lo que hace necesario que se aplique este lema, es que con ayuda de las definiciones que introduce son con las que *nearly_redblack* fue definido.

Como no se sabe si el árbol ***s*** tiene raíz roja o negra, se tienen que probar los dos casos: uno con la hipótesis de que el árbol resultante sea *is_redblack* y otro con *redred_tree*. Estos casos se resuelven sencillamente con la aplicación de alguno de los dos casos de la definición inductiva de *nearly_redblack*, respectivamente.

Instancia de la Función de Inserción

Para poder probar que el resultado de la inserción es una instancia de la clase *redblack*,es decir, que se cumplen con las propiedades descritas por la clase, vamos a necesitar el lema auxiliar que se encuentra en la figura 3.9:

```
Lemma makeBlack_rb {a} `{GHC.Base.Ord a} n t :
nearly_redblack n t -> redblack (makeBlack t).
```

Figura 3.9: Lema *makeBlack_rb*

Este lema auxiliar se resuelve siguiendo la idea de que la definición inductiva *nearly_redblack* únicamente viola las invariantes al poder tener dos nodos rojos consecutivos en su raíz, entonces, si se "pinta" la raíz de negro con la función *makeBlack*, el árbol resultante es un árbol roji-negro le cual respeta la la definición de la clase *redblack*.

La figura 3.10 enuncia la instancia de inserción de la clase *redblack*, la cual también nos generará una demostración:

```
Instance add_rb {a} `{GHC.Base.Ord a} (x:a) (s: RB a) :
redblack s -> redblack (insert x s).
```

Figura 3.10: Instancia de inserción de la clase *redBlack*.

Para poder crear la instancia de la clase *redblack* es necesario usar la definición *insert*, la cual es una envoltura para la función *ins*. Esta función lo que hace es "pintar" la raíz del árbol resultante de la función *ins* de color negro. De esta manera podemos asegurar que el árbol resultante ya no entra en la definición de *redred_tree*.

```
H1 : is_redblack n s
______________________________________(1/1)
redblack (makeBlack (ins x s))
```

En este momento se utiliza el lema auxiliar *makeBlack_rb* el cual nos devuelve lo siguiente:

```
H1 : is_redblack n s
______________________________________(1/1)
nearly_redblack n (ins x s)
```

Esta ultima meta se resuelve aplicando el segundo lema que enunciamos, *ins_arb*, lo cual nos deja con una meta idéntica a la hipótesis H1 y con esto terminamos la verificación de la operación de inserción.

Se puede decir que esta implementación de la función de inserción es correcta respecto a las invariantes establecidas en la definición inductiva *is_redblack*. La operación ha sido verificada formalmente, ahora continuaremos con la función de eliminación.

3.2.3. Verificación de la operación de eliminación

Al igual que en la función de inserción se enuncian lemas para ayudarnos a llegar al resultado de verificar la operación de eliminación. Estos lemas giran en torno a las funciones auxiliares que se usaron para poder demostrar la operación, como *append* y *del*.

A continuación se describe el razonamiento usado para poder verificar dichas funciones.

Primer Lema

La función mas importante para la operación de eliminación es *append*, la cual concatena dos subárboles. Estos dos subárboles son el resultado de buscar, encontrar y eliminar un nodo. En este primer lema se enuncia lo antes descrito: que para cualesquiera dos árboles si estos cumplen con la definición inductiva

```
Lemma append_arb_rb {a} `{GHC.Base.Ord a} (n:nat) (l r: RB a) :
is_redblack n l -> is_redblack n r ->
(nearly_redblack n (append l r)) /\
(notred l -> notred r -> is_redblack n (append l r)).
```

Figura 3.11: Lema *append_arb_rb*.

de *is_redblack*, ambos con altura n, el resultado de concatenar es casi un árbol roji-negro, en otras palabras, la concatenación cumple con la definición de *nearly_redblack*. Pero si los árboles que se van a concatenar además de cumplir con *is_redblack*, también cumplen con *notred*, es decir, las raices de dichos arboles no son rojas, el resultado de concatenar respeta también la definición *is_redblack*. La demostracioón de este lema en *Coq* se describe en seguida:

```
______________________________________(1/1)
Forall (r : RB a) (n : nat),
  is_redblack n l
  -> is_redblack n r
    -> nearly_redblack n (append l r)
      /\ (notred l -> notred r -> is_redblack n (append l r))
```

En este primera etapa de la demostración podemos ver lo que se describió en el párrafo anterior. Se decidió proseguir con esta demostración usando inducción, primero sobre el árbol l y posteriormente sobre r. Los casos base de estas inducciones consisten en simplificación de las expresiones y fácilmente se llega a una hipótesis o a una contradicción. Estos casos no se trataran mas a fondo en este trabajo, nos pasaremos directamente a los casos mas interesantes.

Esta doble inducción nos deja con cuatro casos, expuestos en la figura 3.12, estos son los siguientes:

- Los árboles a concatenar son rojos.
- El árbol que se concatenar a la izquierda es rojo y el derecho es negro.
- El árbol que se concatenar a la izquierda es negro y el derecho es rojo.
- Los árboles a concatenar son negros.

En estos cuatro casos se cuida demasiado el hecho de no desbalancear el árbol, en otras palabras, que la altura negra sea la misma al terminar de la concatenación, es por eso que se tiene cuidado especial en los casos donde se manejan nodos negros, ya que estos son los únicos nodos considerados para el balanceo.

En las siguientes subsecciones explicamos mas a fondo los pasos usados para probar estos casos.

```
______________________________________(1/4)
forall n : nat,
is_redblack n (T R ll lx lr)
-> is_redblack n (T R rl rx rr)
-> nearly_redblack n (append (T R ll lx lr) (T R rl rx rr))
/\ (notred (T R ll lx lr)
-> notred (T R rl rx rr)
-> is_redblack n (append (T R ll lx lr) (T R rl rx rr)))
______________________________________(2/4)
forall n : nat,
is_redblack n (T R ll lx lr)
-> is_redblack n (T B rl rx rr)
-> nearly_redblack n (append (T R ll lx lr) (T B rl rx rr))
/\ (notred (T R ll lx lr)
-> notred (T B rl rx rr)
-> is_redblack n (append (T R ll lx lr) (T B rl rx rr)))
______________________________________(3/4)
forall n : nat,
is_redblack n (T B ll lx lr)
-> is_redblack n (T R rl rx rr)
-> nearly_redblack n (append (T B ll lx lr) (T R rl rx rr))
/\ (notred (T B ll lx lr)
-> notred (T R rl rx rr)
-> is_redblack n (append (T B ll lx lr) (T R rl rx rr)))
______________________________________(4/4)
forall n : nat,
is_redblack n (T B ll lx lr)
-> is_redblack n (T B rl rx rr)
-> nearly_redblack n (append (T B ll lx lr) (T B rl rx rr))
/\ (notred (T B ll lx lr)
-> notred (T B rl rx rr)
-> is_redblack n (append (T B ll lx lr) (T B rl rx rr)))
```

Figura 3.12: Casos del lema *append_arb_rb*.

Concatenación de dos árboles rojos. Este primer caso es la concatenación de dos árboles con raíces rojas, en el siguiente fragmento de la salida del asistente de pruebas se observa como la meta es una conjunción.

```
IHlr : forall (r : RB a) (n : nat),
    is_redblack n lr
     -> is_redblack n r
       -> nearly_redblack n (append lr r)
         /\ (notred lr -> notred r -> is_redblack n (append lr r))
IHrl : forall n : nat,
```

```
    is_redblack n (T R ll lx lr)
    -> is_redblack n rl
      -> nearly_redblack n (append (T R ll lx lr) rl)
        /\ (notred (T R ll lx lr)
           -> notred rl -> is_redblack n (append (T R ll lx lr) rl))
______________________________________(1/1)
forall n : nat,
  is_redblack n (T R ll lx lr)
  -> is_redblack n (T R rl rx rr)
   -> nearly_redblack n (append (T R ll lx lr) (T R rl rx rr))
     /\ (notred (T R ll lx lr)
       -> notred (T R rl rx rr)
        -> is_redblack n (append (T R ll lx lr) (T R rl rx rr)))
```

Podemos observar que la segunda parte de la conjunción es una contradicción, ya que al introducir los antecedentes de la meta tendríamos lo siguiente:

```
H21 : notred (T R ll lx lr)
H22 : notred (T R rl rx rr)
______________________________________(1/1)
is_redblack n (append (T R ll lx lr) (T R rl rx rr))
```

Evidentemente las dos funciones *notred* de las hipótesis $H21$ y $H22$ se evaluan a falso y por esto es una contradicción.

Nos queda por demostrar la primera parte de la conjunción, la meta de este caso, como se ve en seguida, es que al ser concatenados un par de arboles rojos el árbol resultante cumple con la definición de ser de $nearly\_redblack$, es decir, que la raíz del árbol puede tener dos nodos rojos consecutivos.

```
______________________________________(1/2)
nearly_redblack n (append (T R ll lx lr) (T R rl rx rr))
```

El siguiente paso seria simplificar esta expresión, la cual cae en el caso de dos nodos rojos de la función *append* y nos resulta en la siguiente meta:

```
______________________________________(1/1)
redred_tree n
  match append lr rl with
  | T R lr' x rl' => T R (T R ll lx lr') x (T R rl' rx rr)
  | _ => T R ll lx (T R (append lr rl) rx rr)
  end
```

Podemos ver que la caza de patrones depende de la evaluación de la expresión $append(lr, rl)$, sea rbt, esto nos daria dos casos:

- El primer caso, como se ve en seguida, se puede resolver usando las definiciones inductivas de $redred\_tree$ e $is\_redblack$, y las metas resultantes son resultados directos de aplicar las hipótesis que se muestran.

```
H8 : notred lr
H9 : is_redblack n ll
H14 : notred rl
H16 : notred rr
H18 : is_redblack n rr
H19 : nearly_redblack n E
H20 : notred lr -> notred rl -> is_redblack n E
H21 : redred_tree n E
______________________________________(1/2)
redred_tree n (T R ll lx (T R E rx rr))
```

- El segundo caso es un poco mas complejo, pues se tienen que ver los casos en que *rbt*, resulta en un árbol con raíz roja y negra:
  - En caso de que el árbol sea rojo, se aplica de igual manera las definiciones inductivas mencionados en el caso anterior y las metas resultantes son implicaciones directas de las hipótesis que se muestran.

    ```
    H6 : notred ll
    H9 : is_redblack n ll
    H14 : notred rl
    H16 : notred rr
    H20 : notred lr ->
          notred rl -> is_redblack n (T R r1_1 a0 r1_2)
    ______________________________________(1/1)
    redred_tree n (T R (T R ll lx r1_1) a0 (T R r1_2 rx rr))
    ```

  - El caso donde el árbol es negro, al igual que en el caso pasado se hacen uso de las defincionces inductivas ya mencionadas y se siguen directamente de las siguientes hipótesis.

    ```
    H6 : notred ll
    H8 : notred lr
    H9 : is_redblack n ll
    H10 : is_redblack n lr
    H14 : notred rl
    H16 : notred rr
    H17 : is_redblack n rl
    H18 : is_redblack n rr
    H19 : nearly_redblack n (T B r1_1 a0 r1_2)
    H20 : notred lr ->
          notred rl -> is_redblack n (T B r1_1 a0 r1_2)
    ______________________________________(1/1)
    redred_tree n (T R ll lx (T R (T B r1_1 a0 r1_2) rx rr))
    ```

Con este procedimiento queda demostrado este caso de concatenar/unir dos áboles rojos con la función *append*, se puede apreciar como los pasos de la demostración tienden a repetirse, esto puede significar que existan una serie de comandos y/o tácticas del asistente de pruebas que nos ayuden a acortar esta

prueba, sin embargo, en este trabajo se esta tomando el camino mas extenso para mostrar la simplificación de la linea de pensamiento al demostrar estructuras complejas.

Concatenación de un árbol rojo y uno negro. Ahora es turno de analizar la demonstración del caso donde se concatena un árbol rojo y uno negro, es decir, $append(r, b)$, donde r es un árbol rojo y b es uno negro.

```
IHlr : forall (r : RB a) (n : nat),
     is_redblack n lr
     -> is_redblack n r
       -> nearly_redblack n (append lr r)
         /\ (notred lr -> notred r -> is_redblack n (append lr r))
______________________________________(1/1)
forall n : nat,
  is_redblack n (T R ll lx lr)
  -> is_redblack n (T B rl rx rr)
   -> nearly_redblack n (append (T R ll lx lr) (T B rl rx rr))
     /\ (notred (T R ll lx lr)
       -> notred (T B rl rx rr)
        -> is_redblack n (append (T R ll lx lr) (T B rl rx rr)))
```

En este segundo caso la conjunción tambien contiene una contradicción en la mitad derecha de esta, ya que se tiene *notred (T R ll lx lr)*, entonces al igual que el caso pasado solo resolveremos la primera mitad de la conjunción. Para esta demostración tenemos que guiar al asistente de pruebas un poco mas de lo normal, pues le tenemos que decirle que r y el árbol *(T B rl rx rr)* son el mismo.

```
r := T B rl rx rr : RB a
IHlr : forall n : nat,
     is_redblack n lr
     -> is_redblack n r
       -> nearly_redblack n (append lr r)
         /\ (notred lr -> notred r -> is_redblack n (append lr r))
n : nat
H1 : is_redblack n (T R ll lx lr)
H2 : is_redblack n r
______________________________________(1/1)
nearly_redblack n (T R ll lx (append lr r))
```

Podemos observar que se realizo una simplificación de la meta, donde se desarrollo lo mas posible la función append y se introdujeron los antecedentes a las hipótesis. Para poder demostrar esta nueva meta tenemos que destruir la hipótesis 'IHlr', lo cual nos introduciría los dos antecedentes de la misma como metas. Se destruye esta hipótesis para poder obtener su consecuente como hipótesis.

```
r := T B rl rx rr : RB a
IHlr : forall n : nat,
     is_redblack n lr
     -> is_redblack n r
       -> nearly_redblack n (append lr r)
         /\ (notred lr -> notred r -> is_redblack n (append lr r))
IHrl : forall n : nat,
     is_redblack n (T R ll lx lr)
     -> is_redblack n rl
       -> nearly_redblack n (append (T R ll lx lr) rl)
         /\ (notred (T R ll lx lr)
            -> notred rl -> is_redblack n (append (T R ll lx lr) rl))
n : nat
H1 : is_redblack n (T R ll lx lr)
H2 : is_redblack n r
______________________________________(1/3)
is_redblack n lr
______________________________________(2/3)
is_redblack n r
______________________________________(3/3)
nearly_redblack n (T R ll lx (append lr r))
```

Para poder demostrar los dos primeros casos basta con aplicar la definición inductiva *is_redblack*, lo cual nos introduce las hipótesis necesarias para poder cumplir las metas. Para el último caso nos basta de igual manera con aplicar la misma definición inductiva a H1 y a la meta aplicar las definiciones de *nearly_redblack* y *redred_tree* y esto nos da metas, que gracias a las nuevas hipótesis integradas por H1, se pueden probar sin mayor problema.

Con esto demostrado este caso esta completo.

Concatenación de un árbol negro y uno rojo. En este caso se invierten los colores con respecto al caso anterior; el árbol derecho es rojo y el izquierdo es negro. Este caso, al igual que el pasado, requiere de una pequeña ayuda al asistente de pruebas, la cual explicaremos mas adelante.

```
IHlr : forall (r : RB a) (n : nat),
     is_redblack n lr
     -> is_redblack n r
       -> nearly_redblack n (append lr r)
         /\ (notred lr -> notred r -> is_redblack n (append lr r))
IHrl : forall n : nat,
     is_redblack n (T B ll lx lr)
     -> is_redblack n rl
       -> nearly_redblack n (append (T B ll lx lr) rl)
         /\ (notred (T B ll lx lr)
            -> notred rl -> is_redblack n (append (T B ll lx lr) rl))
```

```
______________________________________(1/1)
forall n : nat,
  is_redblack n (T B ll lx lr)
  -> is_redblack n (T R rl rx rr)
    -> nearly_redblack n (append (T B ll lx lr) (T R rl rx rr))
      /\ (notred (T B ll lx lr)
         -> notred (T R rl rx rr)
           -> is_redblack n (append (T B ll lx lr) (T R rl rx rr)))
```

En este caso al igual que los dos pasados, como uno[10] de los arboles es de color rojo, la segunda parte de la conjunción vuelve a ser una contradicción, por la expresión *notred*.

Entonces solo nos quedamos con la primera mitad de la conjunción:

```
l := T B ll lx lr : RB a
IHrl : forall n : nat,
     is_redblack n l
     -> is_redblack n rl
       -> nearly_redblack n (append l rl)
         /\ (notred l -> notred rl -> is_redblack n (append l rl))
n : nat
H1 : is_redblack n l
H2 : is_redblack n (T R rl rx rr)
______________________________________(1/2)
nearly_redblack n (T R (append l rl) rx rr)
```

Podemos apreciar que este caso es el caso espejo del caso pasado, entonces el procedimiento a usar para demostrar esta meta es el mismo, lo único que cambia es cuales hipótesis se usan para lograr esto. En el caso pasado se destruyo la hipótesis IHlr, en este caso se usa su contraparte IHrl y el resto de la demostración se sigue directamente de las nuevas metas introducidas y del uso de las definiciones inductivas mencionadas en el caso pasado.

Concatenación de dos árboles negros. Este último caso es el único que no incluye una contradicción ya que esta se daba al tener un árbol rojo como uno de los dos árboles que se pasan a la función *append*, pero en este caso los dos árboles a concatenar son negros, entonces la conjunción completa será probada.

```
IHlr : forall (r : RB a) (n : nat),
     is_redblack n lr
     -> is_redblack n r
       -> nearly_redblack n (append lr r)
         /\ (notred lr -> notred r -> is_redblack n (append lr r))
IHrl : forall n : nat,
     is_redblack n (T B ll lx lr)
```

[10] o los dos.

```
     -> is_redblack n rl
       -> nearly_redblack n (append (T B ll lx lr) rl)
         /\ (notred (T B ll lx lr)
            -> notred rl -> is_redblack n (append (T B ll lx lr) rl))
______________________________________(1/1)
forall n : nat,
  is_redblack n (T B ll lx lr)
  -> is_redblack n (T B rl rx rr)
    -> nearly_redblack n (append (T B ll lx lr) (T B rl rx rr))
      /\ (notred (T B ll lx lr)
         -> notred (T B rl rx rr)
           -> is_redblack n (append (T B ll lx lr) (T B rl rx rr)))
```

Esta demostración se inicia con una inducción sobre la altura negra, es decir, una inducción sobre n. Esto porque hacer la inducción sobre esta propiedad nos garantiza que el resultado de la concatenación no estará desbalanceado.

Esta inducción nos da el caso base con $n = 0$, seguido de la separación de la conjunción y esto nos da 2 casos base, como se muestra en seguida:

```
IHlr : forall n : nat,
     is_redblack n lr
     -> is_redblack n rl
       -> nearly_redblack n (append lr rl)
         /\ (notred lr -> notred rl -> is_redblack n (append lr rl))
IHrl : forall n : nat,
     is_redblack n (T B ll lx lr)
     -> is_redblack n rl
       -> nearly_redblack n (append (T B ll lx lr) rl)
         /\ (notred (T B ll lx lr)
            -> notred rl -> is_redblack n (append (T B ll lx lr) rl))
H1 : is_redblack 0 (T B ll lx lr)
H2 : is_redblack 0 (T B rl rx rr)
______________________________________(1/2)
nearly_redblack 0 (append (T B ll lx lr) (T B rl rx rr))
______________________________________(2/2)
notred (T B ll lx lr)
-> notred (T B rl rx rr)
  -> is_redblack 0 (append (T B ll lx lr) (T B rl rx rr))
```

Estos casos base se resuelven aplicando las definiciones inductivas correspondientes tanto a las metas como a las hipótesis H1 y H2, esto nos da las hipótesis necesarias para probar estos dos casos.

Nos queda por probar el paso inductivo, en seguida podemos ver que la hipótesis de inducción 'IH' es parte de IHlr, la cual se obtuvo de destruir esa hipótesis, en el siguiente paso se explica porque se decidió destruir esta hipótesis y no su contraparte IHrl.

```
IHlr : forall n : nat,
     is_redblack n lr
     -> is_redblack n rl
       -> nearly_redblack n (append lr rl)
         /\ (notred lr -> notred rl -> is_redblack n (append lr rl))
IHrl : forall n : nat,
     is_redblack n (T B ll lx lr)
     -> is_redblack n rl
       -> nearly_redblack n (append (T B ll lx lr) rl)
         /\ (notred (T B ll lx lr)
            -> notred rl -> is_redblack n (append (T B ll lx lr) rl))
n : nat
H1 : is_redblack (S n) (T B ll lx lr)
H2 : is_redblack (S n) (T B rl rx rr)
IH : nearly_redblack n (append lr rl)
______________________________________(1/1)
nearly_redblack (S n) (append (T B ll lx lr) (T B rl rx rr))
/\ (notred (T B ll lx lr)
   -> notred (T B rl rx rr)
     -> is_redblack (S n) (append (T B ll lx lr) (T B rl rx rr)))
```

Proseguimos con la separación de la conjunción, lo cual nos da dos casos que trabajaremos por separado:

Primera mitad de conjunción

```
______________________________________(1/1)
nearly_redblack (S n) (append (T B ll lx lr) (T B rl rx rr))
```

después de simplificar la meta de este caso, nos queda una meta que depende del resultado de una llamada recursiva a *append* de los subárboles lr y rl, lo cual nos genera otros dos casos:

```
______________________________________(1/2)
is_redblack (S n) (lbalS ll lx (T B E rx rr))
______________________________________(2/2)
is_redblack (S n)
  match c with
  | R => T R (T B ll lx r1) a0 (T B r2 rx rr)
  | B => lbalS ll lx (T B (T c r1 a0 r2) rx rr)
  end
```

Como podemos ver en ambas metas, tenemos una función nueva, $lbalS$, esta es una función de balanceo,la cual extiende a las funciones que ya se habían usado con anterioridad en la función de inserción, como lo son $rbal'$, $rbal$ y $lbal$.

Para poder resolver esta parte de la demostración nos apoyaremos de otro lema, figura 3.13, el cual ilustra una propiedad de la operación $lbalS$.

```
Lemma lbalS_rb {a} `{GHC.Base.Ord a}
(n : nat) (l : RB a) (x : a ) (r : RB a) :
nearly_redblack n l -> is_redblack (S n) r ->
              notred r -> is_redblack (S n) (lbalS l x r).
```

Figura 3.13: Lema $lbalS\_rb$.

Lo que el lema, arriba escrito en sintaxis de *Coq*, quiere decir es que si tenemos un par de arboles, sean l y r, un número natural n y un elemento x, si el árbol l cumple con la definición inductiva $nearly\_redblack$ y r no es de color rojo y cumple con la definición inductiva $is\_redblack$, entonces balancear estos dos arboles con $lbalS$ resulta en un árbol roji-negro que cumple con la definición $is_redblack$.

Ls demostración de este lema se convierte en un análisis de casos en el cual solamente es necesario simplificar, aplicar las definiciones inductivas y las metas que se generan son consecuencias directas de las hipótesis generadas, la inducción no es necesaria.

Regresando a las dos metas generadas por destruir la función append, si nos fijamos en la primera , podemos ver que se puede aplicar directamente el lema $lbalS_rb$, lo cual nos genera 3 nuevas metas:

```
H1 : is_redblack (S n) (T B ll lx lr)
H2 : is_redblack (S n) (T B rl rx rr)
IH : nearly_redblack n E
______________________________________(1/3)
nearly_redblack n ll
______________________________________(2/3)
is_redblack (S n) (T B E rx rr)
______________________________________(3/3)
notred (T B E rx rr)
```

Estas metas de nuevo caen en el caso de simplificar y aplicar las respectivas definiciones inductivas para obtener las metas deseadas, de esta manera el primer caso queda resuelto.

Ahora nos vamos al segundo caso generado al destruir la función *append* el cual nos dice que tenemos que hacer un análisis de casos sobre el color del nodo:

```
H1 : is_redblack (S n) (T B ll lx lr)
H2 : is_redblack (S n) (T B rl rx rr)
IH : nearly_redblack n (T R r1 a0 r2)
______________________________________(1/2)
is_redblack (S n) (T R (T B ll lx r1) a0 (T B r2 rx rr))
______________________________________(2/2)
is_redblack (S n) (lbalS ll lx (T B (T B r1 a0 r2) rx rr))
```

Ese análisis de casos nos da dos metas nuevas, una por color, el primer caso solamente requiere simplificación y aplicación de definiciones inductivas

para obtener las metas deseadas. El segundo caso sigue los mismos pasos con la única diferencia se volver a aplicar el lema *lbalS*.

De esta manera queda demostrada la primera mitad de la conjunción.

Segunda mitad de conjunción

```
______________________________________(1/1)
notred (T B ll lx lr)
-> notred (T B rl rx rr)
  -> is_redblack (S n) (append (T B ll lx lr) (T B rl rx rr))
```

Esta segunda mitad sigue exactamente el mismo procedimiento antes descrito con la única diferencia de que se agregan hipótesis nuevas:

```
H1 : is_redblack (S n) (T B ll lx lr)
H2 : is_redblack (S n) (T B rl rx rr)
IH : nearly_redblack n (append lr rl)
H3 : notred (T B ll lx lr)
H4 : notred (T B rl rx rr)
______________________________________(1/1)
is_redblack (S n)
  match append lr rl with
  | T R lr' x rl' => T R (T B ll lx lr') x (T B rl' rx rr)
  | _ => lbalS ll lx (T B (append lr rl) rx rr)
  end
```

Al hacer el análisis de casos destruyendo la función *append* con los parametros *lr* y *rl*, obtenemos exactamente las mismas metas que en la primera parte de la conjunción y al tener mas hipótesis la demostración se acorta por un par de pasos pero el procedimiento es el mismo.

De esta manera el primer lema queda demostrado, con esta demostración, a pesar de ser larga y tediosa, se puede observar el poder del asistente de pruebas, ya que las demostraciones se reducen a álgebra ecuacional, es decir, tratar de igualar la meta con lo que se tiene como hipótesis, esto se seguirá viendo en las siguientes pruebas.

Segundo Lema

Este siguiente lema utiliza una palabra especial en el lenguaje de *Coq*, 'with', esta palabra es un truco para demostrar dos lemas simultáneamente, el cual es el caso como se ve en la figura 3.14.

Como podemos ver, el lema en si define dos lemas, esto se hace de esta manera porque la demostración de uno de estos lemas depende del otro. De esta manera podemos definir ambos lemas y solo usar una sola prueba.

```
del_arb : forall (a : Type) (H : Base.Eq_ a) (H0 : Base.Ord a)
        (s : RB a) (x : a) (n : nat),
```

```
Lemma del_arb {a} `{GHC.Base.Ord a} (s:RB a) (x:a) (n:nat) :
      is_redblack (S n) s ->
      isblack s -> nearly_redblack n (del x s)
 with del_rb  {a} `{GHC.Base.Ord a} (s:RB a) (x:a) (n:nat) :
      is_redblack n s ->
      notblack s -> is_redblack n (del x s).
```

Figura 3.14: Lema *del_arb*

```
        is_redblack (S n) s ->
        isblack s -> nearly_redblack n (del x s)
del_rb : forall (a : Type) (H : Base.Eq_ a) (H0 : Base.Ord a)
        (s : RB a) (x : a) (n : nat),
        is_redblack n s ->
        notblack s -> is_redblack n (del x s)
______________________________________(1/2)
is_redblack (S n) s -> isblack s -> nearly_redblack n (del x s)
______________________________________(2/2)
is_redblack n s -> notblack s -> is_redblack n (del x s)
```

Lo que el asistente de pruebas esta haciendo es que nos esta integrando al ambiente de hipótesis los dos lemas, de esta manera podemos realizar suposiciones con estos y asi ayudarnos a demostrar los lemas. Realizaremos las pruebas de estos lemas por separado.

Prueba de *del_arb*

```
______________________________________(1/1)
is_redblack (S n) s -> isblack s -> nearly_redblack n (del x s)
```

Este lema enuncia lo siguiente: Sea un árbol s, un elemento x y un número natural n, si s cumple con la definicion inductiva *is_redblack* y s es negro, entonces s cumple con la definición de *nearly_redblack* después de eliminar el elemento x. En otras palabras, si tenemos un árbol con la raíz de color negro, el resultado de eliminar un elemento será un árbol casi rojinegro.

La prueba empieza con una inducción sobre s lo cual nos da las dos metas siguientes:

```
______________________________________(1/2)
forall n : nat, is_redblack (S n) E ->
                isblack E -> nearly_redblack n (del x E)
______________________________________(2/2)
forall n : nat,
  is_redblack (S n) (T c s1 a0 s2) ->
  isblack (T c s1 a0 s2) ->
  nearly_redblack n (del x (T c s1 a0 s2))
```

A primera vista podemos apreciar que la primera meta contiene un antecedente falso, el árbol vacío E no puede ser negro, entonces esta meta es una contradicción. La segunda meta podemos ver que si analizamos los dos casos del color del árbol, el caso rojo es igualmente una contradicción por el mismo antecedente *isblack*. Esto solo nos deja con el caso negro de la segunda meta.

```
IHs1 : forall n : nat, is_redblack (S n) s1 ->
                       isblack s1 -> nearly_redblack n (del x s1)
IHs2 : forall n : nat, is_redblack (S n) s2 ->
                       isblack s2 -> nearly_redblack n (del x s2)
______________________________________(1/1)
forall n : nat,
is_redblack (S n) (T B s1 a0 s2)
-> isblack (T B s1 a0 s2) ->
nearly_redblack n (del x (T B s1 a0 s2))
```

Después de introducir los antecedentes y simplificar la meta, se hace un análisis de casos sobre la operación *del*, primero se ve el caso si el nodo a eliminar esta en el subárbol derecho y después en el izquierdo.

```
IHs1 : forall n : nat, is_redblack (S n) s1 ->
                       isblack s1 -> nearly_redblack n (del x s1)
IHs2 : forall n : nat, is_redblack (S n) s2 ->
                       isblack s2 -> nearly_redblack n (del x s2)
H1 : is_redblack (S n) (T B s1 a0 s2)
H2 : isblack (T B s1 a0 s2)
H6 : is_redblack n s1
H8 : is_redblack n s2
______________________________________(1/2)
nearly_redblack n
match s1 with
| T B _ _ _ => lbalS (del x s1) a0 s2
| _ => T R (del x s1) a0 s2
end
______________________________________(2/2)
nearly_redblack n
(if _GHC.Base.>_ x a0
 then
  match s2 with
  | T B _ _ _ => rbalS s1 a0 (del x s2)
  | _ => T R s1 a0 (del x s2)
  end
 else append s1 s2)
```

Seguimos con el análisis de los dos casos:

- En el primer caso seguimos con la destrucción del árbol $s1$ para tener un análisis de casos, si simplificamos con las definiciones inductivas estas metas, eventualmente encontramos metas de las siguientes formas:

```
is_redblack n (del x E)

is_redblack n (del x (T R s1_1 a1 s1_2))
```

Estos casos son casos particulares del lema *del_rb*, para solucionar esto usamos una táctica de *Coq* llamada *assert*, la cual nos deja agregar hipótesis, las cuales después tendremos que demostrar, en este caso esta quedara demostrada al terminar de demostrar todo este lema. Entonces como en este caso estamos destruyendo $s1$, la hipótesis a agregar seria:

```
IHl' : forall n : nat, is_redblack n s1 ->
                  notblack s1 -> is_redblack n (del x s1)
```

Al agregarla al inicio de la prueba podemos solamente aplicarla cuando lleguemos a los casos arriba mencionados.

- El segundo caso se divide en dos, la primera parte es el caso espejo al pasado, se realiza lo mismo pero para el árbol $s2$, lo cual nos da la siguiente hipótesis a agregar:

```
IHr' : forall n : nat, is_redblack n s2 ->
                  notblack s2 -> is_redblack n (del x s2)
```

Se aplica de la misma manera y llegamos a la segunda parte donde nos resulta la siguiente meta:

```
______________________________________(1/1)
nearly_redblack n (append s1 s2)
```

La cual es un caso particular del lema antes demostrado *append_arb_rb*.

Prueba de *del_rb*

```
______________________________________(1/1)
is_redblack n s -> notblack s -> is_redblack n (del x s)
```

Este segundo lema enuncia lo siguiente: Sea un árbol s, un elemento x y un número natural n, si s cumple con la definicion inductiva *is_redblack* y s no es negro, entonces s cumple con la definición de *is_redblack* después de eliminar el elemento x. En otras palabras, si tenemos un árbol con la raíz de color rojo, el resultado de eliminar un elemento será un árbol casi rojinegro.

El enunciado con respecto al anterior busca que el resultado sea mas especifico, pues la propiedad de ser *is_redblack* es la que buscamos que las operaciones cumplan. Sin embargo la demostración en términos de *Coq* no es muy distinta; iniciamos con inducción sobre s.

```
______________________________________(1/2)
forall n : nat, is_redblack n E ->
           notblack E ->
```

```
            is_redblack n (del x E)
______________________________________(2/2)
forall n : nat, is_redblack n (T c s1 a0 s2) ->
            notblack (T c s1 a0 s2) ->
            is_redblack n (del x (T c s1 a0 s2))
```

La primera meta se soluciona simplificando hasta obtener la meta deseada, mientras que la segunda meta se le hace un análisis sobre el color c, el cual arroja dos casos, rojo y negro. El caso de que c sea negro es una contradicción porque no cumple con la meta de ser *notblack*, solamente nos enfocaremos en el caso en que c es rojo.

```
IHs1 : forall n : nat,
         is_redblack n s1 ->
         notblack s1 -> is_redblack n (del x s1)
IHs2 : forall n : nat,
         is_redblack n s2 ->
         notblack s2 -> is_redblack n (del x s2)
H1 : is_redblack n (T R s1 a0 s2)
H2 : notblack (T R s1 a0 s2)
______________________________________(1/1)
is_redblack n (del x (T R s1 a0 s2))
```

Al igual que en *del_arb* se hacen los casos de si el elemento a eliminar esta en el subárbol derecho o izquierdo. Otra similitud que esta prueba tiene con respecto con la pasada es que también tenemos que agregar una hipótesis extra, en este caso de *del_arb*. De aquí en adelante la prueba es muy similar a la anterior, simplificar, aplicar definiciones inductivas hasta llegar a contradicciones o a las metas deseadas.

En este lema se usan lemas auxiliares muy similares a *lbalS_rb*[11], como su espejo *rbalS_rb* o sus contrapartes *rbalS_arb* y *lbalS_arb*. Estos son lemas de balanceo sencillos de demostrar pero muy largos, tediosos y repetitivos, por lo tanto no se incluirán en este trabajo[12].

Hasta este momento solo se han demostrado partes de la operación total de eliminación, como juntar dos subárboles después de eliminar su raíz, que pasa si eliminamos de un árbol con raíz roja o de uno con raíz negra. En seguida uniremos todos estos lemas en uno.

Instancia de la función de eliminación

```
Instance remove_rb s x : redblack s -> redblack (remove x s).
```

Figura 3.15: Instancia de eliminación de la clase *redblack*.

[11] descrito en la prueba de la función *append*

[12] Se pueden consultar en: `https://github.com/DavidFHCh/Tesis-FTW`

Al igual que en la funci'on de inserción, terminamos la operación de eliminación generando una instancia de la clase *redblack*, figura 3.15, al igual que en la operación opuesta, requerimos de un lema auxiliar con respecto a la clase *redblack*, figura 3.16.

```
Lemma makeBlack_rb {a} `{GHC.Base.Ord a} n t :
nearly_redblack n t -> redblack (makeBlack t).
```

Figura 3.16: Lema *makeBlack_rb*.

Lo que este enunciado describe es la propiedad de que si un árbol t cumple con la definición inductiva de ser *nearly_redblack*, pintar su raíz de color negro lo convierte en una instancia de la clase *redblack*. La desmostración de este lema es muy simple gracias al asistente de pruebas, ya que solo basta con hacer un análisis de casos sobre el árbol t:

- El árbol vacio E, la meta a demostrar para este caso es:

  ```
  nearly_redblack n E -> redblack (makeBlack E)
  ```

  Como la clase *redblack* esconde un existencial en su definición, para poder demostrar este caso basta con decir que existe n con valor 0, esto nos da un caso trivial al ser la misma definición inductiva de *is_redblack*.

- El segundo caso se reduce a los dos casos en los que puede caer la definición inductiva de *nearly_redblack*.

  ```
  H1 : nearly_redblack n (T c t1 a0 t2)
  H2 : is_redblack n (T c t1 a0 t2)
  ______________________________________(1/2)
  redblack (makeBlack (T c t1 a0 t2))
  ```

  Este primer caso se reduce a hacer un análisis de casos sobre el color c, la solución de ambos colores consiste en, decir que existe n' tal que su valor es *S(n)*, despuésde esto se simplifican las expresiones hasta obtener que las metas cumplan con las hipótesis.

  ```
  H1 : nearly_redblack n (T c t1 a0 t2)
  H2 : redred_tree n (T c t1 a0 t2)
  ______________________________________(2/2)
  redblack (makeBlack (T c t1 a0 t2))
  ```

  El segundo caso es mas corto que el primero, ya que al hacer el análisis de los colores, podemos ver que la definición de *redred_tree* no esta definida para árboles negros, entonces solo nos queda demostrar para árbles rojos, sin embargo, los pasos a seguir para este caso son los mismos que para el color rojo del caso anterior.

Con este lema demostrado ya contamos con todas las herramientas para poder demostrar que si tenemos un árbol que es instancia de la clase *redblack* y eliminamos un elemento de el, este sigue siendo instancia de la clase, esta demostración comienza con un análisis de casos sobre el árbol *s*:

```
H1 : is_redblack n E
_____________________________________(1/2)
redblack (makeBlack (del x E))
_____________________________________(2/2)
redblack (makeBlack (del x (T c s1 c0 s2)))
```

Podemos ver que se hace uso de la función *makeBlack*. En la primera meta basta con aplicar el lema *makeBlack_rb*, simplificar y esta se soluciona. En la segunda meta se tiene que hacer otro análisis de casos, esta vez sobre el color:

```
H1 : is_redblack n (T R s1 c0 s2)
_____________________________________(1/2)
redblack (makeBlack (del x (T R s1 c0 s2)))
```

En la primera meta, color rojo, comenzamos por aplicar *makeBlack_rb*, el cual después de simplificar con la defínción inductiva nos resulta en la siguiente meta:

```
H1 : is_redblack n (T R s1 c0 s2)
_____________________________________(1/1)
is_redblack n (del x (T R s1 c0 s2))
```

La cual es un caso particular del lema *del_rb*, nos basta con aplicarlo, simplificar y esta meta queda resuelta. La única meta que nos quedaría por demostrar seria el caso de de la raíz negra:

```
H1 : is_redblack n (T B s1 c0 s2)
_____________________________________(1/1)
redblack (makeBlack (del x (T B s1 c0 s2)))
```

En este caso hacemos un análisis sobre *n*, los dos casos serian 0 y *S(n)*. Para 0 basta con simplificar y la meta se resuelve, pero para *S(n)*, es necesario volver a aplicar el lema *makeBlack_rb*, una vez que hacemos esto, nos queda la siguiente meta:

```
H1 : is_redblack (S n) (T B s1 c0 s2)
_____________________________________(1/1)
nearly_redblack n (del x (T B s1 c0 s2))
```

La cual resulta ser un caso particular del lema *del_arb*, al aplicar este lema y simplificar nuevamente, las metas resultantes quedan resueltas. Podemos ver que las demostraciones cubiertas en este trabajo, en especial en la operación de

eliminación, son muy repetitivas, y solo buscamos hacer que las metas empaten con las hipótesis que tenemos.

Con esta operación demostrada podemos decir que tenemos una estructura correcta respecto a los invariantes descritos por las definiciones inductivas de *is_redblack* con las operaciones de borrado e inserción y de la misma manera que estas dos operaciones son métodos de la clase *redblack*, por lo cual podemos hacer estas operaciones cuantas veces queramos y el resultado seguirá siendo instancia de esta clase.

Capítulo 4

Conclusiones

Como se ha ilustrado a lo largo de este trabajo, lo que buscamos es otra manera de demostrar la corrección de una estructura de datos, en este caso de un árbol roji-negro usando un asistente de pruebas como lo es *Coq* con una biblioteca de tipos y funciones que se tradujeron de Haskell.

Hemos mencionado en repetidas ocasiones que la opción mas tradicional para realizar una prueba de este estilo seria usar lápiz y papel, pero como se ha visto en capítulos anteriores el desarrollo de la prueba puede llegar a generar demasiados casos, esto lo convierte en una tarea muy complicada y tediosa de escribir, y posteriormente de leer y entender por alguien mas. En cambio un asistente de pruebas como lo es *Coq* da herramientas para simplificar esta tarea y logra reducirla a álgebra ecuacional, ya que como se vio en este trabajo, lo único que se busca obtener es que las metas que queremos probar se igualen con alguna de las hipótesis que se tienen, lo cual también tiene sus detrimentos ya que se deja de razonar de manera formal.

Sin embargo, el uso de una herramienta de esta naturaleza por si sola no simplifica del todo este tipo de pruebas, ya que para poder llegar a un escenario donde se pueda desarrollar una demostración primero tenemos que tener claro que es lo que se quiere probar y codificarlo en el lenguaje que la herramienta comprenda.

En la vida real, esto significaría tener un programa escrito en algún lenguaje de programación como Java, Python, Haskell, etc. y traducirlo al lenguaje de la herramienta. Esto requeriría la implentación de un traductor o en su defecto traducir los programas a mano, esta segunda opción siendo una solución no óptima ya que es muy susceptible a errores. En este trabajo se uso el traductor de Haskell a *Coq* llamado *hs-to-coq* [Spector-Zabusky et al., 2017], que aunque nos dio algunas bibliotecas de Haskell traducidas a *Coq*, esta sigue en estado de desarrollo y aunque Haskell comparte el mismo paradigma que el lenguaje de *Coq*, lograr traducir en un 100 % un lenguaje resulta muy complicado ya que este siempre esta evolucionando, en especial si es un lenguaje tan ampliamente usado como lo es Haskell.

Otra restricción que se tiene que establecer es que no todos los programas

escritos en Haskell podrían ser traducidos al lenguaje de *Coq*, este lenguaje a pesar de que entra en la categoría de lenguajes funcionales, este solo acepta funciones totales. Entonces esto introduce otras problemáticas, la traducción un programa de un paradigma imperativo, lógico, etc. a uno funcional y después asegurar que todas las funciones de este son totales.

Supongamos que resolvemos todos estos problemas que se han presentado hasta ahora, es decir, tenemos un programa donde todas sus funciones son totales y se logro traducir correcta y completamente. Ahora se tienen que generar las definiciones inductivas, las cuales te ayudaran a guardar invariantes de tu programa, y con estas escribir los lemas que se buscan probar para poder decir que tu programa ha sido verificado formalmente, lo cual podría tomar el mismo tiempo que tomo traducir todo el programa al lenguaje de la herramienta.

Actualmente en la industria lo que se hace para minimizar los errores en código, es hacer que este pase por una serie de filtros, es decir, que otra persona revise tu código para ayudarte a encontrar defectos, también en ejecutar pruebas ya existentes para asegurar que el nuevo código no introduzca errores a componentes que funcionaban correctamente dentro del programa y que se escriban pruebas que confirmen el correcto funcionamiento del código a introducir. En este momento la idea de poder probar que un programa cualquiera puede ser probado formalmente usando un asistente de pruebas es muy atractiva, ya que un único desarrollador podría desarrollar la prueba y no depender de código ajeno que muestre que su programa es correcto. Sin embargo, esta idea resulta muy poco factible hoy en día, ya que además de los problemas expuestos con anterioridad (las traducciones del código implementado) se le suma el hecho de que se tendrían que traducir y probar todas las bibliotecas ocupadas del lenguaje que se esta usando, esto antes de pensar en probar tu programa.

Otro acercamiento para poder probar este tipo de programas en la industria seria desarrollar la mayor parte de estos en el asistente *Coq*, realizando esto con las herramientas que su lenguaje nos provee, de esta manera se pueden realizar las demostraciones pertinentes y utilizar la funcionalidad que este posee para extraer código en otros lenguajes, después de haber realizado la prueba, como lo son Haskell y O'Caml. Sin embargo, esto solo nos permitiría desarrollar programas correctos con las funcionalidades que el lenguaje de *Coq* nos ofrezca.

Retomando el punto anterior, otra solución seria desarrollar y probar partes clave de los programas a crear, es decir, módulos pequeños como lo serian las estructuras de datos a usar, como lo podrian ser los árboles roji-negros, listas doblemente ligadas, colas, pilas u otros tipos de árboles. Una vez implementados estos módulos se podrían usar en cualquier parte de código, el problema con hacer esto es que dependiendo del lenguaje al que se extraiga el programa probado, puede ser contraproducente para el desempeño del mismo. Este degradado en el desempeño se puede dar por razones ajenas al código y mas por asuntos relacionados a la implementación del compilador que se usará para generar código ejecutable y que tan optimizado es el mismo. Por ejemplo, el lenguaje C es conocido por ser muy usado en programas que requieren ser muy rápidos en sus operaciones.

Otro problema es que como no todo el código estaría probado formalmente,

para componentes mas grandes se necesitaría caer nuevamente en hacer otro tipo de pruebas, como las unitarias, y como ya hemos mencionado, estas no nos aseguran que los programas sean correctos o completos y por lo tanto, nuestros programas solo estarían parcialmente verificados formalmente.

Como podemos apreciar el demostrar programas con un asistente de pruebas, no es el procedimiento mas amigable hoy en día, sin embargo, si se sigue con la actual trayectoria en el desarrollo de herramientas de traducción como lo es *hs-to-coq*, eventualmente la industria podría comenzar a utilizar herramientas de este estilo para mejorar la calidad de sus productos. Mientras tanto, se tendrán que seguir desarrollando pruebas unitarias de mejor calidad y lograr generar programas que tiendan a la corrección y completud.

Bibliografía

[Appel, 2018] Appel, A. W. (2018). Software foundations, volume 3, verified functional algorithms. [En linea; Consultado el 18 de enero de 2020.].

[Appel and Letouzey, 2011] Appel, A. W. and Letouzey, P. (2011). Msetrbt : Implementation of msetinterface via red-black trees. [En linea; Consultado el 18 de enero de 2020].

[Castéran and Sozeau, 2016] Castéran, P. and Sozeau, M. (2016). A gentle introduction to type classes and relations in coq. [En linea; Consultado el 12 de enero de 2020].

[Geuvers, 2009] Geuvers, H. (2009). Proof assistants: History, ideas and future. *Sadhana*, 34:3–25.

[HaskellWiki, 2020] HaskellWiki (2020). Currying — haskellwiki,. [En linea; Consultado el 21 de julio de 2020].

[López Campos, 2015] López Campos, G. (2015). *Implementaciones funcionales de árboles roji-negros*. TESIUNAM. `http://tesis.unam.mx/` [Consultado el 12 de enero de 2020].

[Okasaki, 1998] Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press, USA.

[Osherove, 2014] Osherove, R. (2014). *The Art of Unit Testing, Second Edition*. Manning Publications. `https://piazza.com/class_profile/get_resource/j11t8bsxngk3r3/j2lw6zcyt5t6lu` [En linea; Consultado el 30 de septiembre de 2020].

[Pelaez, 2019] Pelaez, C. (2019). Material del curso de estructuras de datos. [En linea; Consultado el 6 de octubre de 2020].

[Spector-Zabusky et al., 2017] Spector-Zabusky, A., Breitner, J., Rizkallah, C., and Weirich, S. (2017). Total haskell is reasonable coq. *CoRR*, abs/1711.09286.