Representación de Estructuras de Datos con Invariantes en Haskell

Marco Nicolás Rodríguez Alvariza

Proyecto de Grado
Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República
Orientadores: Alberto Pardo, Marcos Viera
2022

Resumen

Las estructuras de datos utilizadas en programación se caracterizan por mantener un conjunto de invariantes. Dependiendo del lenguaje y paradigma de programación utilizados, estos invariantes son verificados en tiempo de ejecución o compilación. En este proyecto de grado se presenta el problema de implementar invariantes de distintas estructuras de datos en el lenguaje Haskell, utilizando las herramientas que el lenguaje provee para programar a nivel de tipos. La verificación de estos invariantes se realiza en tiempo de compilación. Se implementan árboles binarios de búsqueda (BST) y árboles balanceados (AVL). Ambos son implementados siguiendo tres enfoques diferentes, dependiendo de la forma en que se implementan los invariantes. Se realiza una comparación de estos enfoques desde distintos ángulos: desempeño, modularidad del código, etc.

Índice

Capítulo 1

Introducción

En programación, las estructuras de datos se caracterizan por tener al menos un invariante de algún tipo. Las listas, por ejemplo, tienen un largo asociado; los árboles binarios de búsqueda mantienen un orden en las claves de sus nodos, etc. Estos invariantes imponen restricciones a las operaciones que se aplican sobre las estructuras. Insertar un elemento en una lista debe incrementar su largo en una unidad. Al programar una estructura de datos se debe garantizar que sus operaciones respeten en todo momento estos invariantes. No es admisible que la inserción en una lista devuelva otra con largo menor a la original.

Sea *List* el tipo de las listas de largo arbitrario. La firma, en Haskell, de una función que inserta un elemento sería entonces

$$insert :: List \to List$$

El tipo de *insert* solamente asegura que la función opera sobre una lista y que retorna otra. Es responsabilidad del programador asegurar que su implementación respeta el invariante del largo. Se necesita una verificación posterior a la implementación.

Si en lugar de tener un único tipo para todas las listas, se tuviera un tipo $List\ n$, para las listas de largo n (es decir, una familia de tipos indexada en el largo), entonces el invariante aparece explícitamente en la firma de *insert*:

$$insert :: List\ n \to List\ (n + 1)$$

Si ahora el programa compila correctamente, entonces se puede asegurar que la implementación de *insert* cumple con la firma. Esto implica que respeta el invariante. Bajo este enfoque, es el compilador (más específicamente el *type checker*), y no el programador, el que asegura que se respeta el invariante de la estructura.

Para invariantes más complejos, surge la pregunta de cómo verificarlos. Para verificar el orden de los elementos en una lista existen al menos dos posibilidades. La primera consiste en definir una estructura auxiliar que empaquete (a modo de *wrapper*) la lista original y que sea esta estructura auxiliar la que verifique el orden. La segunda posibilidad es incorporar el invariante al proceso de construc-

ción de las listas, es decir solo permitir la construcción de listas ordenadas. Al primer enfoque se lo denominará *enfoque externalista* y al segundo *internalista*.

En el enfoque externalista existen a su vez dos variantes. Dado que la verificación es realizada por una estructura de datos diferente, las operaciones del tipo de dato (e.g., listas) se definen sobre una estructura sin restricciones. Se puede verificar los invariantes luego de haber realizado varias operaciones en serie o inmediatamente después de cada operación. Al primer enfoque se lo denominará *completamente externalista* para diferenciarlo del segundo (simplemente, *externalista*).

Siguiendo un enfoque similar al mencionado, por ejemplo en [LM13] se muestra una implementación en el lenguaje Haskell de vectores ordenados, de modo que sea el compilador el que certifique que los vectores se encuentran ordenados.

El objetivo del presente proyecto, fuertemente inspirado en [LM13], es mostrar que para otras estructuras de datos también es posible incorporar los invariantes en su tipo, de modo de delegar al compilador la tarea de asegurar que todas las operaciones respetan los invariantes.

En el Capítulo 2 se presentan las herramientas que provee el lenguaje Haskell para incorporar invariantes a nivel de tipos. Se presenta el patrón de *singletons* como mecanismo (necesario en Haskell) para vincular las ejecuciones a nivel de valores con las computaciones a nivel de tipos. También se muestran herramientas para la implementación de proposiciones y pruebas en Haskell vinculadas a la verificación de los invariantes. Todo lo anterior requiere la utilización de extensiones del lenguaje que agregan funcionalidades y brindan mayor control sobre el proceso de compilación.

En el Capítulo 3 se presenta una implementación de árboles balanceados (*AVL*) que no hace uso de programación a nivel de tipos. Se utiliza como punto de partida para el análisis de las implementaciones que se presentan en los capítulos subsiguientes. También se muestran y enuncian formalmente las propiedades que deben ser demostradas para asegurar que los árboles cumplen efectivamente con la condición de ser *AVL*. En los capítulos siguientes se muestra la implementación de estas propiedades.

En el Capítulo 4 se implementan los árboles binarios de búsqueda (*BST*), incorporando el invariante de orden de las claves a nivel de tipos. Se implementan siguiendo los enfoques externalistas e internalista. Se compara el desempeño de cada enfoque.

En el Capítulo 5 se extienden las implementaciones de árboles *BST* de cada enfoque para incorporar el invariante de árbol *AVL*. Se discute la flexibilidad de cada enfoque para la incorporación de nuevos invariantes y la reutilización de código.

En el Capítulo 6 se presentan las conclusiones.

En los Apéndices se encuentran los resultados completos de las pruebas de rendimiento (benchmarks), junto con representaciones gráficas y las especificaciones del hardware y software utilizados.

El código fuente se encuentra disponible en un repositorio público `https://github.com/nico-rodriguez/type-safe-avl` y como librería en el reposiorio Hackage `https://hackage.haskell.org/package/type-safe-avl-1.0.0.0`.

Capítulo 2

Preliminares

En este capítulo se presentan las herramientas necesarias para programar a nivel de tipos e implementar pruebas en Haskell. Se presentan las extensiones del lenguaje, que agregan funcionalidades y brindan mayor control sobre el proceso de compilación. Se introduce los *singletons* como mecanismo para programar a nivel de tipos. Por último, se muestran herramientas para la implementación de proposiciones y pruebas en Haskell. A lo largo del capítulo se presentan funciones y estructuras de datos útiles para programar a nivel de tipos, que ya se encuentran implementadas en bibliotecas externas.

2.1 Lenguaje y herramientas

Haskell es un lenguaje de programación funcional fuertemente tipado. Existen varios compiladores para el lenguaje: *Glasgow Haskell Compiler (GHC)*, *Utrecht Haskell Compiler (UHC)*, *LLVM Haskell Compiler (LHC)*. Con el paso de los años, *ghc* se convirtió en el compilador estandar de hecho.

El compilador utilizado es *ghc*[21], que permite la utilización de extensiones del lenguaje. Desde el punto de vista del lenguaje, estas extensiones implican nuevas opciones de sintaxis (e.g., promoción de tipos, funciones a nivel de tipos y *GADTs*); desde la perspectiva del programador, supone la posibilidad de modificar el proceso de compilación, en particular la etapa de *type checking* [MP12, *2.1 Code metrics, 2.3 Compiling Haskell code*].

2.2 *Data kinds*

Los valores son los elementos que se computan dinámicamente (en tiempo de ejecución); mientras que los tipos proveen información estática (en tiempo de compilación) sobre los valores.

Además en Haskell existen los *kinds*, que informalmente son los "tipos de los tipos". Es decir, los *kinds* son a los tipos como los tipos son a los valores. Todos los tipos primitivos (`Int`, `Bool`, `String`, etc) tienen *kind* `Type`. En Haskell, en principio, la gramática de los *kinds*[Mar20a] es muy sencilla:

```
k ::= Type | k → k
```

Código 2.1: Lenguaje de los *kinds*.

Sin embargo, la extensión del lenguaje *Data Kinds*[20a] permite extender el conjunto de los *kinds*, mediante la promoción de los constructores de datos a constructores de tipos [Yor+12].

Por ejemplo, el constructor de números naturales

```
{-# LANGUAGE DataKinds #-}

data Nat = Z | S Nat
```

Código 2.2: Definición de los números naturales.

es promovido a un constructor de tipos de *kind* `Nat`. El tipo `Nat` es promovido al *kind* `Nat` y los valores `Z`, `S Z` a los tipos `'Z`, `'S 'Z`, etc.

La comilla `'` se utiliza como un mecanismo de desambiguación. Puesto que Haskell permite que los constructores de tipos y datos tengan el mismo nombre, como por ejemplo

```
data T = T Int
```

Código 2.3: Constructor de tipos y de datos con el mismo nombre.

Luego, cuando `T` aparece en un tipo, ¿se refiere al tipo `T` (de *kind* `Type`) o al constructor de datos promovido (de *kind* `Type → T`)?. En ese caso, `'T` hace referencia al constructor de datos promovido.

Adicionalmente, la extensión *Data Kinds* habilita la utilización de literales[20b] numéricos (de *kind* `Nat`) y alfabéticos (de *kind* `Symbol`) a nivel de tipos, así como algunas de sus operaciones básicas. Por ejemplo

```
{-# LANGUAGE DataKinds #-}

import GHC.TypeNats
import Data.Word
import Foreign

newtype ArrPtr (n :: Nat) a = ArrPtr (Ptr a)

-- literal 4096 como parte del tipo
clearPage :: ArrPtr 4096 Word8 → IO ()
clearPage (ArrPtr p) = ...
```

Código 2.4: Literales numéricos a nivel de tipos.

Para el caso particular de los naturales a nivel de tipos, la biblioteca `GHC.TypeNats`[lib21a] provee una implementación del *kind* `Nat`, así como las operaciones básicas sobre naturales: suma, resta, división, módulo, etc.

2.3 *GADTs*

Los *Generalised Algebraic Data Types (GADTs)* son otra extensión del lenguaje[20c]. Permiten omitir la restricción de que los constructores de datos deben siempre retornar un tipo de dato simple.

Por ejemplo, la siguiente es una implementación de listas de elementos de tipo `a` sin *GADTs*

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

Código 2.5: Implementación de listas sin *GADTs*.

Ambos constructores, `Nil` y `Cons`, siempre retornan una lista de tipo `List a`:

$$Nil :: List\ a$$
$$Cons :: a \to List\ a \to List\ a$$

Al omitir esta restricción utilizando la extensión de *GADTs*, es posible indexar el tipo de la lista con su largo, utilizando los naturales `Nat` definidos previamente:

```
data ListNat → Type → Type where
  Nil :: List 'Z a
  Cons :: a → List n a → List ('S n) a
```

Código 2.6: Listas indexadas por su largo a nivel de tipos definidas como *GADTs*.

Los nuevos constructores `Nil` y `Cons` retornan valores de tipos distintos. El tipo `List n a` contiene información del largo de la lista, con la que se pueden verificar propiedades de la misma en tiempo de compilación. Por ejemplo, se puede asegurar que la lista no está vacía para poder tomar su primer elemento

```
head :: List ('S n) a → a
head (Cons x xs) = x
```

Código 2.7: Obtener el primer elemento de una lista. El tipo indica que la lista tiene al menos un elemento.

Una invocación a `head` con una lista vacía resulta en un error en tiempo de compilación. El tipo de una lista vacía es `List 'Z a`, mientras que `head` solo recibe como argumento listas de tipo `List ('S n) a`. Por esta misma razón, no es necesario implementar el comportamiento de `head` en el caso de una lista vacía. La función `head` implementada de esta forma es total. Agregar la definición para el caso de la lista vacía también resultaría en un error de compilación, por la misma razón que antes.

2.4 *Type families*

Las *type families*[20d] son una extensión del lenguaje que permite definir funciones a nivel de tipos. Por ejemplo, para los naturales a nivel de tipos de *kind* `Nat`, es posible definir la suma como

```
type family (m::Nat) + (n::Nat)::Nat where
  'Z     + n = n
  ('S m) + n = 'S (m + n)
```

Código 2.8: Suma de naturales a nivel de tipos.

Luego se puede utilizar este operador como parte de la especificación del tipo de una función. Por ejemplo, la siguiente especificación de concatenación de dos listas

```
append :: List m x → List n x → List (m + n) x
```

Código 2.9: Concatenación de listas indexadas por su largo.

restringe las posibles implementaciones de la función: la lista que `append` retorna debe tener un largo igual a la suma de los largos de las listas que recibe como argumentos. De lo contrario, ocurre un error de compilación, ya que el tipo de la lista que se da como resultado de la función no coincide con el tipo especificado (porque sus largos serían distintos). Además, la definición de la suma a nivel de tipos es recursiva en el primer parámetro, por lo que la implementación de `append` queda restringida a ser recursiva en la primera lista.

2.4.1 Restricciones para la terminación

Es posible definir una *type family* recursiva o incluso una pareja de *type families* mutuamente recursivas. Esto trae aparejado el problema de la terminación de la *type family*. En efecto, en la fase de *type checking* puede ser necesario reducir todas las aplicaciones de *type families* a una forma normal (e.g., reducir la expresión `'Z + 'S 'Z` a `'S 'Z`). Para garantizar la terminación de la fase de *type checking*, se impone un conjunto de restricciones en la definición de las *type families* [Sch+08, Definition 5 (Relaxed Conditions)]: dada una *type family*

```
type family F t1 ... tn where
  F t1 ... tn = t
  ...
```

en toda definición `F t1...tn = t` y en toda aplicación de una *type family* `G s1...sm` en `t` se requiere que:

- `s1...sm` no contengan aplicaciones de *type families*,
- el número de símbolos (constructores de datos y variables de tipo) en `s1...sm` sea estrictamente menor que en `t1...tn`, y
- para cada variable de tipo, el número de veces que ocurre en `s1...sm` no sea mayor al número de veces que ocurre en `t1...tn`.

Estas restricciones aseguran la terminación de la etapa de *type checking* a costa de un menor poder expresivo.

La extensión del lenguaje *Undecidable Instances*[20e] indica al compilador que no verifique estas restricciones. Esto amplía las posibles definiciones de *type families*, en particular para la definición de *type families* mutuamente recursivas. Pero la responsabilidad de asegurar la terminación recae en el programador.

2.5 *Type classes*

La *type class* es una herramienta de Haskell que permite la implementación de polimorfismo ad-hoc, también conocido como sobrecarga. Esto es cuando un valor o función es capaz de adoptar distintos tipos porque posee una definición específica para cada uno de esos tipos.

La definición de una *type class* se divide en dos partes. Una primera parte consiste en la declaración, una interfaz definida como un conjunto de funciones que deben ser implementadas por cualquier tipo que sea instancia de la *type class*; esta implementación es la segunda parte de la definición y se dice que es una instancia de la *type class*[Mar20b].

La biblioteca `base` provee algunas *type classes* con instancias predefinidas. Por ejemplo, la *type class* `Eq`[lib21h], que provee métodos para realizar comparación de valores.

```
-- Minimal complete definition: either '==' or '/='.
--
class  Eq a  where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool

  x /= y = not (x == y)
  x == y = not (x /= y)
```

Código 2.10: Declaración de la *type class* `Eq`.

Cada instancia de `Eq` se define para un tipo `a` implementando la operación `(==)`. Por ejemplo, para la siguiente definición de árboles binarios se define una instancia de `Eq`:

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

instance (Eq a) => Eq (BinTree a) where
  Leaf a1       == Leaf a2       = a1 == a2
  (Node l1 r1) == (Node l2 r2) = (l1 == l2) && (r1 == r2)
  _             == _             = False
```

Código 2.11: Árboles binarios. Instancia de `Eq` para `BinTree`.

`Eq a` se denomina restricción de clase. Restringe los posibles tipos de la variable `a` a aquellos que implementan una instancia de `Eq`. Durante la compilación se activa el mecanismo de *resolución de instancias*[20f], que realiza la búsqueda de las instancias utilizadas en las restricciones de clase.

Las restricciones se definen dentro de lo que se denomina *contexto* de la instancia. Tanto las declaraciones de *type classes* como sus instancias aceptan

contextos con restricciones.

El operador `==` en `a1 == a2` es la igualdad definida en la instancia `Eq a`. Su tipo es `(==)::a → a → Bool`, mientras que el tipo de la igualdad sobre árboles es `(==)::(Eq a) ⟹ BinTree a → BinTree a → Bool`. Omitir la restricción de clase `Eq a` o utilizar un tipo `a` que no implemente la instancia `Eq` produce un error en tiempo de compilación.

El símbolo `_` representa que para ese argumento se admite cualquier valor posible.

2.5.1 Declaración de *type classes*

En principio, las *type classes* solo pueden definirse sobre una única variable de tipo. Esta restricción puede omitirse con la extensión *Multi-Parameter Type Classes*[20g] para definir *type classes* sobre dos o más variables de tipo. Por ejemplo:

```
class Balanceable' (t :: Tree) (us :: US) where
  type Balance' (t :: Tree) (us :: US) :: Tree
  balance' :: ITree t → Proxy us → ITree (Balance' t us)
```

Código 2.12: *Type class* con dos parámetros.

En este caso, la implementación de la función polimórfica `balance'` depende de los tipos `t` y `us`. Para seleccionar la implementación adecuada es necesario indicar ambos tipos en las instancias de la *type class*.

2.5.2 *Scoped type variables*

En principio, el alcance de las variables de tipo de las *type classes* y sus instancias, solo alcanza a su contexto y las signaturas de tipo de sus miembros; no cubre las definiciones[20h]. En el ejemplo siguiente

```
class C a where
  op :: [a] → a
```

Código 2.13: Ejemplo de *type class* con una sola función.

la variable de tipo `a`, declarada en la cabeza de la *type class*, es la misma que aparece en la signatura de tipo de `op`. Sin embargo, en la instancia siguiente

```
instance C b ⇒ C [b] where
  op xs = reverse (head (xs :: [[b]]))
```

Código 2.14: Ejemplo de instancia de *type class* con definición.

la variable de tipo `b` en la expresión `(xs::[[b]])` es una variable nueva que no tiene relación con la variable `b` que aparece en la cabeza de la instancia.

La extensión del lenguaje *Scoped Type Variables* modifica el alcance de las variables de tipo[20i], extendiéndolo al cuerpo de definición de cada método de la *type class*. Con esta extensión, la expresión `(xs::[[b]])` no declara una nueva

variable de tipo, sino que hace referencia a la variable `b` declarada en la cabeza de la instancia.

2.5.3 Terminación de instancias

Anteriormente se presentaron las reglas que restringen definición de *type families* para asegurar la terminación en su evaluación. Análogamente existen reglas que restringen la declaración de instancias y *type classes*, para asegurar la terminación del proceso de resolución de instancias, denominadas *Condiciones de Paterson* [Sul+06, Definition 11 (Paterson Conditions)]. En particular, respecto a las restricciones de clase en los contextos de instancias

```
instance (C' s1 ... sm, ...) => C t1 ... tn where
```

Código 2.15: Esquema general de una instancia de *type class*.

las reglas son las siguientes:

1. Ninguna variable de tipo ocurre más veces en el contexto que en la cabeza de la instancia `C t1...tn`.

2. Cada restricción de clase tiene menos constructores y variables de tipo (contando repeticiones) que la cabeza de la instancia.

3. No se utilizan *type families* dentro de las restricciones de clase. La aplicación de una *type family* puede en principio generar un tipo de tamaño arbitrario, por lo que se rechazan en general.

Estas restricciones aseguran que el proceso de resolución de instancias termine, puesto que cada restricción de clase involucra menos variables que la *cabeza* de la instancia correspondiente [Sul+06].

La tercera restricción puede ser omitida con la extensión *FlexibleInstances*[20j].

Respecto a las otras dos restricciones, estas pueden ser desactivadas mediante la extensión *Undecidable Instances* (también utilizada con *type families*)[20k].

Al igual que con las *type families*, omitir estas restricciones brinda mayor poder expresivo, pero la terminación de la fase de *type checking* pasa a ser responsabilidad del programador.

2.6 Mensajes de error

Dentro de la lógica de un programa pueden ocurrir situaciones de excepción que alteren su flujo normal. Por ejemplo, cuando se busca una clave que no existe en un árbol. Dependiendo del caso, cuando se programa a nivel de valores, puede ser obligado a abortar su ejecución; cuando se programa a nivel de tipos, lo que se aborta es la compilación.

En ambos casos, es recomendable implementar mensajes de error que informen la causa del mismo. En el contexto de programación a nivel de tipos, una

parte de la biblioteca *GHC.TypeLits* ofrece herramientas para abortar la compilación y mostrar un mensaje de error en determinados casos[lib21i].

Tomando como ejemplo la división de números naturales a nivel de tipos

```
type family Div (a::Nat) (b::Nat) :: Nat where
  Div a b = ...
  Div a 0 = TypeError (Text "The number " :<>: ShowType a :<>:
    Text " is not divisible by 0")
```

Código 2.16: Mensaje de error a nivel de tipos en el caso de división por cero.

en el caso de división por cero se maneja mostrando un mensaje acorde a esta situación. Si por ejemplo, se utiliza la *type family* `Div` como `Div 2 0`, se aborta la compilación y se presenta el mensaje de error "`The number 2 is not divisible by 0`".

La *type family* `TypeError` construye el mensaje de error a partir de un tipo de *kind* `ErrorMessage` que puede ser construido mediante

- la aplicación de `Text` a una cadena de caracteres a nivel de tipos,
- la aplicación de `ShowType` a un tipo (o variable de tipo) o
- la concatenación de dos mensajes de error mediante `:<>:`.

La implementación de mensajes de error permite apuntar directamente a la fuente del error. Por otro lado, capturar errores en tiempo de compilación implica una mayor robustez del programa durante su ejecución.

2.7 *Singletons*

Los *singletons* son una clase especial de tipos. Son los miembros de una familia de tipos en la que cada uno tiene un único valor posible (y además cada valor tiene un solo tipo posible). Luego el término *"singleton"* puede designar tanto al tipo como al valor.

Esta biyección entre valores y tipos es lo que permite en Haskell conectar el mundo de los valores con el de los tipos. Pero al mismo tiempo esta biyección implica la duplicación de código a nivel de tipos y de valores.

Por ejemplo, al trabajar con *singletons* de números naturales

```
{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE TypeOperators       #-}
{-# LANGUAGE StandaloneDeriving #-}


import Data.Kind (Type)
import Prelude hiding ((+))

-- Nat se promueve a un constructor de tipos.
data Nat = Z | S Nat
```

```
-- Singletons de los naturales.
-- SNat tiene kind Nat → Type.
data SNat :: Nat → Type where
  SZ :: SNat 'Z
  SS :: SNat n → SNat ('S n)

-- Suma a nivel de tipos.
type family (+) (n :: Nat) (m :: Nat) :: Nat where
  'Z     + n = n
  ('S m) + n = 'S (m + n)

-- Suma a nivel de valores.
(+) :: SNat m → SNat n → SNat (m + n)
SZ     + n = n
(SS m) + n = SS (m + n)
```

Código 2.17: Singletons de números naturales.

en la familia de tipos `SNat n`, cada natural a nivel de tipos `n` se corresponde con un único valor: `'Z` se corresponde con `SZ`, `'S 'Z` se corresponde con `SS SZ`, etc. Para implementar la suma, esta se define sobre los *GADTs* y sobre los tipos de *kind* `Nat`.

Algunos trabajos [EW12] han intentado dar una solución al problema de duplicación de código a nivel de tipos y valores. La biblioteca *singletons*[ES21] permite reducir esta redundancia aunque en este trabajo no se utiliza. A partir de la implementación a nivel de valores de los datos y sus operaciones, la biblioteca genera el código correspondiente a los *singletons* y las operaciones a nivel de tipos, utilizando otra extensión del lenguaje, *Template Haskell*[20l], que permite metaprogramación en tiempo de compilación [SP02]. En el ejemplo anterior, a partir de las implementaciones de `Nat` y `(+)` (suma a nivel de valores), la biblioteca genera el código correspondiente a los *singletons* y la suma a nivel de tipos, es decir `SNat` y la *type family* `(+)`, respectivamente.

2.7.1 Ejemplo de *singletons*

Un ejemplo de uso de *singletons* se presenta en [LM13, *2. A variety of quantifiers*] con la función `chop`, que divide una lista (vector para los autores) en dos partes:

```
chop::SNat m → List (m + n) x → (List m x, List n x)
chop SZ     xs          = (Nil,       xs)
chop (SS m) (Cons x xs) = (Cons x ys, zs)
  where (ys,zs) = chop m xs
```

Código 2.18: Dividir una lista indexada por su largo en dos partes.

El primer parámetro es un `SNat` del cual se conoce, a partir de su tipo `SNat m`, que su valor representa el natural `m`. En este caso es necesario conocer `m` en tiempo de ejecución para determinar el lugar correcto en donde separar la lista. El segundo parámetro es la lista que se va a dividir en dos. Su largo está especificado en el tipo como `m + n`, una suma a nivel de tipos.

Por otro lado, tener `m` a nivel de tipos previene algunos errores de implementación. Esto se debe a que cualquier implementación de `chop` debe ser coherente con su tipo. Por ejemplo, la siguiente implementación del caso recursivo

```
chop (SS m) (Cons x xs) = (ys, zs)
  where (ys,zs) = chop m xs
```

Código 2.19: Error en la implementación de `chop`. Genera un error de compilación.

genera un error en tiempo de compilación. En el ejemplo, no se inserta el elemento `x` al inicio de la lista `ys`. El tipo de `ys` es `List m x`, mientras que `chop` debe retornar una lista de tipo `List (S m) x` en la primera componente de la dupla.

2.8 *Proxies*

En algunos casos solamente se necesita información a nivel de tipos. En esos casos no es necesario definir un *singleton*, porque no se realiza ninguna computación a nivel de valores. Sin embargo, la información a nivel de tipos debe ser transmitida a través de algún valor. Es en estos casos que los *proxies* son útiles.

Por ejemplo, la siguiente definición de la función `take` [LM13, *2. A variety of quantifiers*], que toma los primeros `n` elementos de una lista, es incorrecta:

```
take :: SNat m → List (m + n) x → List m x
take SZ     xs          = Nil
take (SS m) (Cons x xs) = Cons x (take m xs)
```

Código 2.20: Tomar los primeros `m` elementos de una lista. El compilador no puede inferir el tipo `n` por sí solo.

El problema es que el compilador no es capaz de inferir el tipo de `n`. Es necesario proveer explícitamente el tipo `n`, aunque su correspondiente a nivel de valores no sea utilizado.

Para este tipo de situaciones, Haskell provee la biblioteca `Data.Proxy`, la cual define el tipo de dato `Proxy`[lib21b]:

```
data Proxy t = Proxy
```

Código 2.21: Definición de `Proxy`.

La única utilidad del *proxy* es proveer evidencia de tipos cuando no se utiliza el valor correspondiente.

Utilizando *proxies* se agrega `n` al tipo de la función `take`:

```
take :: SNat m → Proxy n → List (m + n) x → List m x
take SZ     _ xs          = Nil
take (SS m) n (Cons x xs) = Cons x (take m n xs)
```

Código 2.22: Tomar los primeros `m` elementos de una lista. `n` no es necesario a nivel de valores pero sí a nivel de tipos.

La definición de `Proxy` se apoya en la extensión del lenguaje *Poly Kinds*[20m] [Yor+12], que habilita al compilador a inferir el *kind* más general posible para cada tipo. Sin esta extensión, el *kind* que se infiere para `Proxy` es `Type → Type`; *Poly Kinds* permite generalizarlo a `forall k. k → Type`. De este modo, puede utilizarse el mismo constructor para definir *proxies* de *kinds* diferentes (por ejemplo, de naturales de *kind* `Nat`). Sin *Poly Kinds*, habría que definir un constructor de *proxies* por cada natural:

```
{-# LANGUAGE DataKinds        #-}
{-# LANGUAGE GADTs            #-}
{-# LANGUAGE KindSignatures #-}

data Nat = Z | S Nat

data Proxy0 :: Nat → Type where
  Proxy0 :: Proxy0 'Z

data Proxy1 :: Nat → Type where
  Proxy1 :: Proxy1 ('S 'Z)

-- etc.
```

Código 2.23: Ejemplos de *proxies* sin *Poly Kinds*.

Es posible definir manualmente un *proxy* como `proxy = undefined::a`, donde `a` es el tipo que se desea hacer explícito en tiempo de compilación. `undefined` es un término que al ser evaluado corta la ejecución de un programa y es un habitante de cualquier tipo (i.e., la expresión `undefined::a` compila correctamente para cualquier tipo `a`).

La función del *proxy* es simplemente aportar información de tipos. Su valor concreto es irrelevante y en general nunca es evaluado. Aún así, evaluar el tipo de dato `Proxy` definido en `Data.Proxy` no aborta la ejecución del programa, como sí ocurre al evaluar `undefined`. En este sentido, `Proxy` es más seguro de usar.

2.9 Proposiciones y pruebas en Haskell

Existe una correspondencia entre el mundo de la programación y el mundo de la lógica que vincula términos y tipos con pruebas y proposiciones lógicas.

En 1958, H.B. Curry mostró que los axiomas del cálculo proposicional se corresponden con los tipos de los combinadores S, K e I de la lógica combinatoria [CFC58]:

Cálculo proposicional	Lógica combinatoria
$(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$	$S\ x\ y\ z = x\ z\ (y\ z)$
$\alpha \to \beta \to \alpha$	$K\ x\ y = x$
$\alpha \to \alpha$	$I\ x = x$

Además de esta relación sintáctica, en 1965, W. Tait muestra que la eliminación de lemas en pruebas (*cut-elimination*) se corresponde con la β-reducción

del cálculo lambda [Tai65]. En otras palabras, puesto que el cálculo lambda de A. Church [Chu36] es la formalización de un modelo de computación (equivalente a las máquinas de Turing [Tur36]), la simplificación de pruebas se corresponde con la evaluación de programas.

Posteriormente, en 1968, W. A. Howard vincula los trabajos de Curry y Tait, y muestra una correspondencia entre la deducción natural y el cálculo lambda simplemente tipado, donde las fórmulas lógicas son a los tipos como las pruebas a los términos lambda; además hizo explícita la relación entre simplificación de pruebas y computación de términos lambda (i.e., reducción a forma normal) [How80].

Relaciones similares existen entre otros tipos de lógicas y modelos de computación [Wad15]. Pero en líneas generales estas correspondencias pueden resumirse en tres grandes puntos

1. proposiciones como tipos
2. pruebas como programas
3. simplificación de pruebas como evaluación de programas

Los puntos 1 y 2 establecen una biyección entre lógica y programación. Al agregar el punto 3, la biyección se convierte en un isomorfismo, porque preserva la estructura interna de pruebas y programas, de simplificación y evaluación [Wad15]. Esta noción de *Proposiciones como Tipos* se conoce como *Isomorfismo de Curry-Howard*, aunque también es referida por otros nombres en la literatura [Wad15].

A continuación se presentan las herramientas de programación a nivel de tipos actualmente disponibles en Haskell, que permiten implementar demostraciones de proposiciones sobre invariantes de estructuras de datos.

2.9.1 Igualdad de tipos

Las pruebas en Haskell, como se verá más adelante, se implementarán como proposiciones sobre igualdad de tipos. Se hará uso del módulo `Data.Type.Equality`[lib21c] de la biblioteca `base`[lib21d], que viene por defecto con el compilador *ghc*.

El término de prueba que provee la evidencia para la igualdad de tipos, definido en `Data.Type.Equality`, se implementa como un *GADT*:

```
data a :~: b where
  Refl :: a :~: a
```

Código 2.24: Término de prueba del módulo `Data.Type.Equality` de la biblioteca `base`. Provee la evidencia de la igualdad de los tipos `a` y `b`.

Los términos de tipo `a :~: b` solo pueden ser construidos mediante el constructor `Refl`, cuyo tipo es `a :~: a`. Esto implica que la única forma de construir un término de tipo `a :~: b` es cuando el compilador puede verificar que los tipos `a` y `b` son iguales.

Por otro lado, al realizar *pattern matching*[Mar20c] y exponer el término `Refl`[lib21e], se incorpora al contexto la igualdad de tipos `a` $\sim$ `b`; luego el compilador puede hacer uso de ella para inferir los tipos de otros términos que involucren a `a` o `b`.

2.9.2 Conversión de tipos

Con el término de prueba de la igualdad proposicional, se implementa la conversión segura de tipos mediante la función `castWith`[lib21f]:

```
castWith :: (a :~: b) → a → b
castWith Refl x = x
```

Código 2.25: Conversión segura (*type-safe cast*) de tipos mediante igualdad proposicional.

Al realizar *pattern matching* con el término `Refl` en el argumento de tipo `a` :$\sim$: `b`, se incorpora al contexto, dentro del alcance del *pattern match*, la igualdad de tipos `a` $\sim$ `b` que necesita el compilador para retornar el valor `x` (originalmente de tipo `a`) como un valor de tipo `b`.

La función de `castWith` se generaliza a `gcastWith`[lib21g]:

```
gcastWith :: (a :~: b) → ((a ~ b) => r) → r
gcastWith Refl x = x
```

Código 2.26: Generalización de la conversión segura (*type-safe cast*) de tipos mediante igualdad proposicional.

La notación `(a` $\sim$ `b) => r` significa que para poder verificar el tipo `r`, se necesita tener en el contexto la igualdad `a` $\sim$ `b`.

Dados la evidencia de que los tipos `a` y `b` son iguales (primer argumento) y un valor de tipo `r` que necesita en el contexto la información de que `a` $\sim$ `b` (segundo argumento), la función `gcastWith` retorna el valor de tipo `r`.

En el contexto de demostrar proposiciones en Haskell, esta función se utiliza para incluir una demostración dentro de otra: si se tiene una prueba de `a` :$\sim$: `b` y una prueba parcial de `r`, en la que resta demostrar que `a` $\sim$ `b`, entonces la función `gcastWith` construye una prueba de `r` a partir de ambas.

2.9.3 Términos de prueba en Haskell

A partir de la siguiente definición de suma de naturales a nivel de tipos

```
type family (+) (m::Nat) (n::Nat) :: Nat where
  m + 'Z     = m              -- (1)
  m + ('S n) = 'S (m + n)     -- (2)
```

Código 2.27: Suma de naturales `Nat` a nivel de tipos. Se define mediante recursión en el segundo argumento.

un ejemplo de término de prueba es el siguiente:

```
plusRightId :: (a + 'Z) :~: a
plusRightId = Refl
```

Código 2.28: El término `plusRightId` demuestra a nivel de tipos: $\forall n \in Nat,\ n + 0 = n$.

En el proceso de compilación, al momento de decidir si el tipo de `plusRightId` es correcto, el *type checker* reduce la expresión `a + 'Z` a la expresión `a`, en virtud de la definición *(1)* de la *type family* `(+)`[20n]. Por último, verifica que `a :~: a` es un tipo válido para asignar a `Refl`.

El ejemplo anterior fue sencillo de implementar porque la propia definición de la suma implementa la propiedad del neutro a la derecha. Sin embargo, para probar el neutro a la izquierda

```
plusLeftId :: ('Z + a) :~: a
```

Código 2.29: Representación a nivel de tipos de la propiedad del neutro a la izquierda.

es necesario construir la prueba por inducción en los naturales `Nat`. El caso base es resuelto por el compilador igual que en el ejemplo anterior. Para el caso inductivo, es necesario manejar el valor de `a` en tiempo de ejecución a fin de poder referirse a la hipótesis inductiva. Para esto se utilizan los *singletons* de naturales `Nat`.

El objetivo es demostrar `'Z + ('S n) ~ 'S n`. Aplicando la definición *(2)* de la *type family* `(+)`, el compilador puede simplificar la igualdad:

$$
\begin{aligned}
&{}'Z + ('S\ n) \sim {}'S\ n \\
\rightsquigarrow\ &{}'S\ ('Z + n) \sim {}'S\ n \\
\rightsquigarrow\ &{}'Z + n \sim n
\end{aligned}
$$

La última igualdad es exactamente la hipótesis inductiva. El término `plusLeftId n::'Z + a:~:a` construye la hipótesis inductiva, que es aplicada mediante `gcastWith`:

```
plusLeftId :: SNat a → ('Z + a) :~: a
-- Caso base: 'Z + 'Z ~ 'Z
plusLeftId SZ     = Refl
-- Caso inductivo: 'Z + a ~ a => 'Z + ('S a) ~ 'S a
plusLeftId (SS n) = gcastWith (plusLeftId n) Refl
```

Código 2.30: El término `plusLeftId` demuestra a nivel de tipos: $\forall n \in Nat,\ 0 + n = n$.

2.9.4 Uso de términos de prueba en Haskell

Un ejemplo en donde es útil el uso de pruebas es en la concatenación de listas indexadas por su largo.

```
data List :: Nat → Type → Type where
  Nil :: List 'Z a
  Cons :: a → List n a → List ('S n) a
```

Código 2.31: Listas indexadas por su largo a nivel de tipos.

Retomando la función `append` mencionada anteriormente

```
append :: List m x → List n x → List (m + n) x
```

Código 2.32: Función de concatenación de listas con información del largo a nivel de tipos.

gracias al uso de los términos de prueba `plusRightId` y `sumSucc` (ver más adelante), es posible dar una implementación de esta función:

```
append :: SNat m → SNat n → List m a → List n a → List (m + n) a
append SZ      n Nil          ys = gcastWith (plusLeftId n) ys
append (SS m)  n (Cons x xs) ys = gcastWith (sumSucc m n) $ Cons x (
    append m n xs ys)
```

Código 2.33: Concatenación de listas indexadas por su largo. Se agregan los argumentos `SNat m` y `SNat n` para construir los términos de prueba.

Es necesario agregar los argumentos `SNat m` y `SNat n` para tener una representación a nivel de valores de `m` y `n` a fin de construir los términos de prueba `plusLeftId n` y `sumSucc m n`.

`sumSucc` se implementa de forma similar a `plusRightId`

```
sumSucc :: SNat x → SNat y → ('S x + y) :~: 'S (x + y)
sumSucc x SZ     = Refl
sumSucc x (SS y) = gcastWith (sumSucc x y) Refl
```

Código 2.34: El término `sumSucc` demuestra: $\forall x, y \in Nat,\ (x{+}1){+}y = (x{+}y){+}1$.

En el caso que la primera lista sea vacía, `append` retorna directamente la segunda lista, de tipo `List m a`. Pero el tipo del resultado es `List ('Z + n) a`. Se utiliza `plusLeftId` para mostrar al compilador que `'Z + n` $\sim$ `n`.

En cambio, si la primera lista no sea vacía, se devuelve la lista `Cons x (append m n xs ys)`, cuyo tipo es `List ('S (m + n)) a`, mientras que el tipo del resultado de `append` es `List (('S m) + n) a`. El término de prueba `sumSucc` muestra que `'S (m + n)` $\sim$ `('S m) + n`.

2.9.5 La lógica del sistema de tipos de Haskell

En la Sección 2.8 se mostró el uso de *proxies* para proveer evidencia a nivel de tipos a través de un término `Proxy` que no contiene información a nivel de valores. Un detalle que no se mencionó, es que no hay restricciones a los tipos que se pueden introducir mediante *proxies*.

```
p = Proxy :: Proxy ('S 'Z :~: 'Z)
```

Código 2.35: Introducción de una proposición falsa a través de `Proxy`.

Algo similar ocurre con el término `undefined`, mencionado en la misma sección.

```
p = undefined :: ('S 'Z :~: 'Z)
```

Código 2.36: Introducción de una proposición falsa a través de `undefined`.

De esta forma el sistema de tipos de Haskell, visto como sistema lógico, permite introducir cualquier proposición (incluso proposiciones falsas). En otras palabras, es un sistema lógico inconsistente.

El uso de las funciones `castWith` y `gcastWith` limita el uso de `undefined`. Estas funciones (ver implementación en la Sección 2.9.2) evalúa los términos de prueba al realizar *pattern matching* con el término `Refl`. En otras palabras, estas funciones verifican que las pruebas no estén definidas mediante `undefined`. En tiempo de compilación tal vez no se detecte la falsedad de la proposición, pero en tiempo de ejecución se aborta el programa.

Para poder utilizar el sistema de tipos como entorno de verificación, dado que es un sistema lógico inconsistente, se debe prestar especial atención de no introducir ningún tipo asociado a una proposición falsa. En particular, en este trabajo no se utiliza `undefined` en ningún momento y los *proxies* no introducen información nueva al sistema de tipos.

Capítulo 3

Implementación no segura de árboles *AVL*

En este capítulo se presenta una implementación tradicional de árboles *AVL*, que no hace uso de programación a nivel de tipos; toda la información se implementa a nivel de valores. Por esta razón, el compilador no puede ser utilizado como certificador de la corrección de esta implementación, como sí ocurre con las implementaciones que se presentan en los próximos capítulos.

El árbol `AVL` implementado es homogéneo. Los valores que almacena deben ser todos de un mismo tipo, aunque este tipo es arbitrario y queda a elección del usuario. Sobre esta estructura de datos se implementan las tres operaciones básicas: inserción, búsqueda y borrado.

También se muestra y enuncian formalmente las propiedades que deben ser demostradas para asegurar que los árboles cumplen efectivamente con la condición de ser *AVL*. En los capítulos siguientes se muestra la implementación de estas propiedades.

Finalmente, para evaluar el desempeño de esta implementación, se toma como referencia el desempeño de una implementación similar (no segura) de árboles `BST` Por simplicidad, no se presenta la implementación no segura de árboles `BST`.

3.1 Árboles AVL

Los árboles *AVL* son árboles binarios de búsqueda que mantienen en cada subárbol una condición particular de balance en las alturas.

Definición 3.1.1 (*BST*)**.** Se dice que un árbol binario `t` es un *árbol binario de búsqueda (Binary Search Tree)*, si para cada nodo `n` se cumplen

- las claves del subárbol izquierdo de `n` son menores que la clave de `n`, y
- las claves del subárbol derecho de `n` son mayores que la de `n`.

Observación. Para simplificar la implementación, no se admiten claves repetidas.

Sobre árboles *BST* con n nodos, el proceso de búsqueda se realiza en tiempo $O(n)$ en el peor caso [Knu98, *6.2.2. Binary Tree Searching*]. Imponer restricciones en las alturas permite optimizar el tiempo de las búsquedas. La restricción en las alturas, de *Adelson-Velsky y Landis* [AL62], reduce la altura global del árbol, mejorando el tiempo de búsqueda a $O(\log n)$ en el peor caso [Knu98, *6.2.3. Balanced Trees*].

Definición 3.1.2 (*Árbol balanceado*)**.** Se dice que un árbol binario es un *árbol balanceado*, si para cada nodo se cumple que la diferencia de alturas de sus subárboles es a lo sumo 1.

Definición 3.1.3 (*AVL*)**.** Se dice que un árbol binario es *AVL*, si es *BST* y es un *árbol balanceado*.

3.2 Constructores

Se implementa cada nodo con una clave, representada por un número natural, y un valor, que puede ser de cualquier tipo de *kind* `Type`.

```
data Node :: Type → Type where
  Node :: Show a => Int → a → Node a
```

Código 3.1: Constructor de nodos para árboles `AVL` no seguros.

Para la clave, se utiliza el tipo primitivo `Int` por simplicidad. También podría utilizarse el tipo `Nat` definido en el capítulo previo.

En cuanto al tipo del valor almacenado en el nodo, su tipo es visible en el tipo del nodo. Por otro lado, se pide la instancia `Show a` para poder transformar el nodo a una cadena de caracteres (y posteriormente mostrarlo al usuario).

Para construir árboles `AVL` se definen dos constructores: uno construye el árbol vacío; el otro, un nuevo árbol a partir de un nodo y dos *AVL*:

```
data AVL :: Type → Type where
  E :: AVL a
  F :: AVL a → Node a → AVL a → AVL a
```

Código 3.2: Constructor no seguro de árboles `AVL`.

El único invariante presente en el *GADT* es el de árbol binario. En los capítulos siguientes se muestra como incluir los otros dos invariantes (orden en las claves y balance en las alturas) con técnicas de programación a nivel de tipos.

Los árboles construidos tienen tipo `AVL a`, donde a es el tipo de los valores almacenados. En el constructor `F` se observa que este tipo coincide con el del valor almacenado en el nodo. El *GADT* `AVL` es polimórfico, ya que permite construir árboles de tipos diferentes, e.g.: `AVL Char`, `AVL Int`, etc. Además, el tipo está parametrizado por el tipo de los valores almacenados. En otras palabras, el tipo de estos valores determina el tipo del árbol. Esta forma de polimorfismo se conoce como *polimorfismo paramétrico*.

3.3 Operaciones

Las operaciones, y su análisis, que se presentan a continuación se basan en [Knu98, *6.2.3. Balanced Trees*].

3.3.1 Rotaciones

Las operación de inserción puede desbalancear un árbol `AVL`. Dependiendo de las alturas de sus subárboles, este árbol puede encontrarse inicialmente en tres configuraciones diferentes, pero solamente a partir de dos de ellas existe la posibilidad de que termine desbalanceado luego de la inserción:

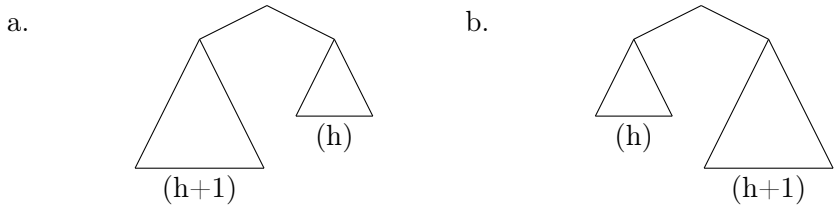


Figura 3.1: Configuraciones iniciales de árboles `AVL` a partir de las que se puede generar un desbalance.

Los estados en que puede quedar el árbol luego de una inserción, se clasifican en cuatro configuraciones:

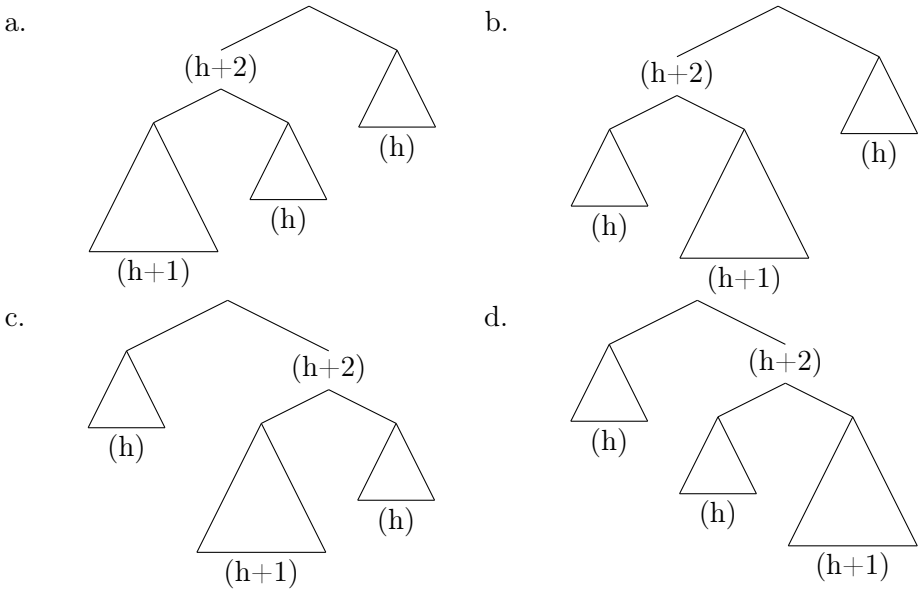


Figura 3.2: En los casos a y b, el árbol se encuentra desbalanceado hacia la izquierda (configuración inicial a), mientras que en c y d hacia la derecha (configuración inicial b). En ningún caso el árbol cumple la condición de equilibrio en la raíz: la diferencia en las alturas de los subárboles de la raíz es $(h+2)-h=2$.

En el caso del borrado, pueden ocurrir dos configuraciones adicionales, cuando el subárbol con altura máxima está balanceado:

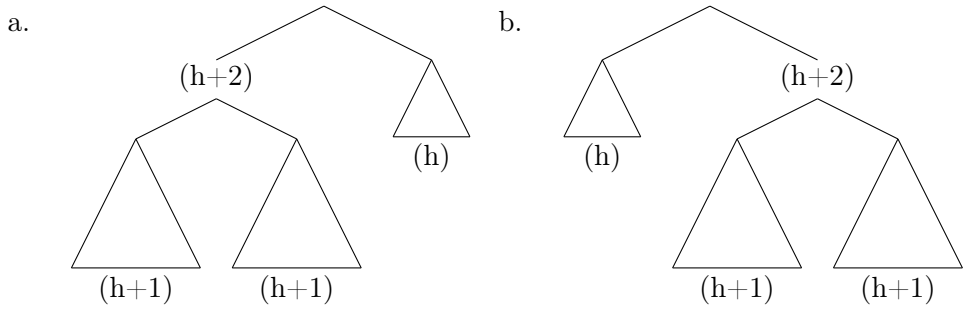


Figura 3.3: a. El árbol se encuentra desbalanceado en el subárbol izquierdo, y este está balanceado; b. Idem para el subárbol derecho.

En resumen, cuando se produce un desbalance en un árbol `AVL` luego de una inserción o borrado, este se identifica mediante

1. una diferencia de 2 entre las ramas izquierda y derecha, y
2. la diferencia en las alturas de los subárboles de la rama de mayor altura, que puede ser $+1$ o -1 en el caso de la inserción, o, adicionalmente, 0 en el caso del borrado.

Para verificar el primer punto, se implementa el tipo de dato `US` y la función `unbalancedState`. Los valores `LeftUnbalanced` y `RightUnbalanced` indican una diferencia de 2 entre las ramas izquierda y derecha, y cuál es la de mayor altura; `NotUnbalanced` indica que las alturas de las ramas están balanceadas. La función `unbalancedState` se aplica a las alturas de cada rama y retorna el valor de `US` correspondiente:

```
data US = LeftUnbalanced | RightUnbalanced | NotUnbalanced

unbalancedState :: Int → Int → US
unbalancedState 0 0   = NotUnbalanced
unbalancedState 1 0   = NotUnbalanced
unbalancedState 0 1   = NotUnbalanced
unbalancedState 2 0   = LeftUnbalanced
unbalancedState 0 2   = RightUnbalanced
unbalancedState h1 h2 = unbalancedState (h1−1) (h2−1)
```

Código 3.3: Averiguar si hay una diferencia de 2 en las alturas de las ramas izquierda y derecha, y cual es la rama con mayor altura.

Una implementación más eficiente consiste en analizar directamente la diferencia `h1 - h2` y retornar el valor de `US` apropiado de acuerdo al resultado. Sin embargo, la implementación elegida es más coherente con la correspondiente a los enfoques seguros del Capítulo 5.

El segundo punto se verifica análogamente, mediante el tipo de dato `BS` y la función `balancedState`, aplicados a la rama de mayor altura.

```
data BS = LeftHeavy | RightHeavy | Balanced

balancedState :: Int → Int → BS
balancedState 0 0   = Balanced
balancedState 1 0   = LeftHeavy
balancedState 0 1   = RightHeavy
balancedState h1 h2 = balancedState (h1−1) (h2−1)
```

Código 3.4: Averiguar cual es el subárbol con mayor altura.

En función de los valores de `US` y `BS` se determina en cual de las seis configuraciones se encuentra el árbol. Por ejemplo, en la Figura 3.2, el caso a. se identifica con `(LeftUnbalanced,LeftHeavy)`: la invocación a `unbalancedState` sobre el árbol devuelve `LeftUnbalanced` y la de `balancedState` sobre el subárbol izquierdo retorna `LeftHeavy`; el caso b. se identifica con `(LeftUnbalanced,RightHeavy)`, etc.

Las configuraciones pueden ocurrir en un nodo interno del árbol, no necesariamente en su nodo raíz. En cualquier caso, el balance entre las alturas se recupera aplicando *rotaciones* sobre los nodos en donde ocurre el desbalance.

Las rotaciones a aplicar dependen de la configuración en la que se encuentre el árbol:

```
rotate :: AlmostAVL a → US → BS → AVL a
-- | Left−Left case (Right rotation)
rotate (FF (F ll lnode lr) node r) LeftUnbalanced LeftHeavy = F ll lnode (F lr node r)
rotate (FF (F ll lnode lr) node r) LeftUnbalanced Balanced  = F ll lnode (F lr node r)
-- | Right−Right case (Left rotation)
rotate (FF l node (F rl rnode rr)) RightUnbalanced RightHeavy = F (F l node rl) rnode rr
rotate (FF l node (F rl rnode rr)) RightUnbalanced Balanced   = F (F l node rl) rnode rr
-- | Left−Right case (First left rotation, then right rotation)
rotate (FF (F ll lnode (F lrl lrnode lrr)) node r) LeftUnbalanced RightHeavy =
  F (F ll lnode lrl) lrnode (F lrr node r)
-- | Right−Left case (First right rotation, then left rotation)
rotate (FF l node (F (F rll rlnode rlr) rnode rr)) RightUnbalanced LeftHeavy =
  F (F l node rll) rlnode (F rlr rnode rr)
```

Código 3.5: Implementación de las rotaciones sobre las seis configuraciones.

donde `AlmostAVL` es un tipo de dato que representa a los árboles que se encuentran desbalanceados luego de una inserción o borrado

```
data AlmostAVL :: Type → Type where
  FF :: AVL a → Node a → AVL a → AlmostAVL a
```

Código 3.6: Constructor de árboles `AlmostAVL`.

Se utiliza el tipo `AlmostAVL` para hacer referencia a este tipo de árboles en los teoremas posteriores y para que la implementación no segura sea coherente con la implementación segura.

La diferencia entre `AVL` y `AlmostAVL` es que las alturas de los subárboles izquierdo y derecho, del nodo raíz, en un `AlmostAVL` no se encuentran balanceadas necesariamente.

Definición 3.3.1 (*Árbol casi balanceado*)**.** Un árbol binario no vacío `t = FF l (Node x v) r` se dice *casi balanceado* si se cumple que `l` y `r` son *árboles balanceados*.

Definición 3.3.2 (*Almost AVL*)**.** Un árbol binario no vacío `t = FF l (Node x v) r` se dice *Almost AVL* (*Casi AVL*) si se cumplen

- `l` y `r` son *AVL*
- `t` es *BST*.

Observación. Todo *AVL* es también *Almost AVL*. El recíproco no se cumple.

Observación. Las dos definiciones anteriores son más generales de lo que se necesita. Los árboles *casi balanceados* que se manejan en la implementación son los que resultan de una inserción o borrado. Estos presentan una configuración como en las Figuras 3.2 o 3.3.

Para simplificar el código, las rotaciones no son invocadas directamente, sino a través de las operaciones `balance` y `balance'`:

```
balance :: AlmostAVL a → AVL a
balance t@(FF l _ r) =
  balance' t (unbalancedState (height l) (height r))

balance' :: AlmostAVL a → US → AVL a
balance' (FF l n r)              NotUnbalanced   = F l n r
balance' t@(FF (F ll _ lr) _ _) LeftUnbalanced  =
  rotate t LeftUnbalanced $ balancedState (height ll) (height lr)
balance' t@(FF _ _ (F rl _ rr)) RightUnbalanced =
  rotate t RightUnbalanced $ balancedState (height rl) (height rr)
```

Código 3.7: Balancear un árbol.

El tipo de `balance` brinda información sobre su implementación. Observando solamente la firma de tipo `balance::AlmostAVL → AVL`, se deduce que `balance` se aplica a un árbol parcialmente desbalanceado para que recupere el balance.

En general, `balance` no puede rebalancear cualquier árbol *casi AVL*. Por ejemplo, no puede rebalancear un árbol en donde la diferencia entre las ramas izquierda y derecha es mayor a 2 (e.g.: rama izquierda con altura 30 y rama derecha con altura 4). Sin embargo, como `balance` se aplica luego de una inserción o borrado sobre un árbol *AVL*, el árbol se encuentra en una de las configuraciones de las Figuras 3.2 o 3.3. En estos casos, `balance` sí recupera el balance en las alturas.

De todas formas, notar que los constructores de `AVL` y `AlmostAVL` permiten construir árboles binarios que no son *AVL* ni *Almost AVL*. Tampoco se tiene ninguna garantía, a nivel de implementación, que `balance` retorne un *AVL* a

partir de un *Almost AVL*. Ni siquiera se garantiza que las claves permanezcan ordenadas.

Por lo tanto, para poder garantizar que la implementación de `balance` respeta el significado de su firma de tipo, es necesario demostrar formalmente los siguientes teoremas:

Teorema 3.3.1. Para todo árbol binario `t`, si `t` es *BST*, entonces `balance t` también es *BST*.

Teorema 3.3.2. Para todo árbol binario `t`, si `t` es un árbol *casi balanceado* con una configuración como en las Figuras 3.2 o 3.3, entonces `balance t` es un árbol *balanceado*.

El primer enunciado establece que la operación `balance` mantiene el orden de las claves, mientras que el segundo establece que permite recuperar el balance en las alturas. De ambos teoremas se deduce el siguiente:

Teorema 3.3.3. Para todo árbol binario `t`, si `t` es un árbol *casi AVL* con una configuración como en las Figuras 3.2 o 3.3, entonces `balance t` es un árbol *AVL*.

La demostración de estos teoremas y de los que se enunciarán en secciones siguientes quedan fuera del alcance de este trabajo.

3.3.2 Inserción

La operación de inserción (así como la de búsqueda y borrado) es recursiva y recorre el árbol teniendo en cuenta el orden de las claves: cada llamada recursiva se realiza sobre la rama izquierda si la clave a insertar/buscar/borrar es menor a la clave del nodo raíz actual; en caso que sea mayor, se llama recursivamente sobre el subárbol derecho. Recordar que la implementación no admite claves repetidas.

La comparación entre la clave a insertar y la clave del nodo del árbol se realiza con la función `compare`, que retorna uno de los valores `LT`, `EQ` o `GT` como resultado de la comparación. Este valor se utiliza en la llamada recursiva como tercer parámetro de `insert'`.

```
insertAVL :: Show a => Int → a → AVL a → AVL a
insertAVL x  v  E                    = F E (Node x v) E
insertAVL x' v' t@(F _ (Node x _) _) =
  insertAVL' (Node x' v') t (compare x' x)

insertAVL' :: Node a → AVL a → Ordering → AVL a
insertAVL' node (F l _ r) EQ = F l node r
insertAVL' n'   (F E n r) LT = balance (FF (F E n' E) n r)
insertAVL' n'@(Node x _) (F l@(F _ (Node ln _) _) n r) LT =
  balance $ FF (insertAVL' n' l (compare x ln)) n r
insertAVL' n' (F l n E) GT = balance (FF l n (F E n' E))
insertAVL' n'@(Node x _) (F l n r@(F _ (Node rn _) _)) GT =
  balance $ FF l n (insertAVL' n' r (compare x rn))
```

Código 3.8: Inserción no segura.

Notar la aplicación del constructor `FF`, del tipo de dato `AlmostAVL`, en los casos en que el árbol resulta potencialmente desbalanceado.

Para simplificar las operaciones, no se admiten claves repetidas. La inserción actualiza el valor que se encontraba en el árbol, si la clave que se inserta ya estaba presente.

Luego de insertar el nuevo nodo, se invoca a `balance` para rebalancear el nuevo árbol (potencialmente desbalanceado).

Análogamente al caso de las rotaciones, luego de insertar un nuevo nodo es necesario probar que el árbol resultante mantiene los invariantes de orden en las claves y alturas equilibradas:

Teorema 3.3.4. Si `x::Int`, `a::a` y `t::BST a` es *BST*, entonces `insertAVL x a t` también es *BST*.

Teorema 3.3.5. Si `x::Int`, `a::a` y `t::AVL a` es un arbol *balanceado*, entonces `insertAVL x a t` también es un árbol *balanceado*.

La demostración de estos teoremas puede valerse de lemas como el siguiente:

Lema 3.3.1. Si `n::Int`, `x::Int`, `a` es un valor de *kind* `Type`, `l` es *BST* y se cumplen

- las claves de `l` son menores que `n`
- $x < n$

entonces las claves de `insert x a l` también son menores que `n`.

Si se demostrara el Teorema 3.3.4 mediante inducción en la estructura del árbol, este lema es útil para el caso en que la inserción se realiza a la izquierda.

Además, como `insert` hace uso de `balance`, las demostraciones pueden apoyarse en los Teoremas 3.3.1 y 3.3.2, correspondientes a `balance`.

En la Sección 4.6 se muestra una implementación del Teorema 3.3.4 y el Lema 3.3.1.

3.3.3 Búsqueda

La búsqueda también es recursiva y sigue el mismo esquema que la inserción:

```
lookupAVL :: Int → AVL a → Maybe a
lookupAVL _ E                    = Nothing
lookupAVL x t@(F _ (Node n _) _) = lookupAVL' x t (compare x n)

lookupAVL' :: Int → AVL a → Ordering → Maybe a
lookupAVL' _ E                                  _  = Nothing
lookupAVL' _ (F _ (Nnode _ a)                  EQ = Just a
lookupAVL' _ (F E _ _)                         LT = Nothing
lookupAVL' _ (F _ _ E)                         GT = Nothing
lookupAVL' x (F l@(F _ (Node ln _) _) _ _) LT =
  lookupAVL' x l (compare x ln)
lookupAVL' x (F _ _ r@(F _ (Node rn _) _)) GT =
  lookupAVL' x r (compare x rn)
```

Código 3.9: Búsqueda no segura.

Como esta operación no modifica el árbol, no existe riesgo de modificar los invariantes del árbol. Sin embargo, se presenta otro problema: ¿qué ocurre si la clave buscada no se encuentra en el árbol?

Para resolver este problema, se utiliza el tipo `Maybe`:

```
data Maybe a = Just a | Nothing
  deriving (Eq, Ord)
```

Código 3.10: Tipo `Maybe`.

Si se encuentra la clave, se retorna su valor asociado como `Just a`; en caso que la clave no se encuentre en el árbol, se retorna `Nothing`.

Utilizando programación a nivel de tipos es posible implementar una búsqueda segura, que detecta en tiempo de compilación si la clave se encuentra en el árbol. Si el programa compila correctamente, se tiene la certeza de que la búsqueda encuentra la clave. En esa situación, solo es necesario implementar los casos en que la búsqueda es exitosa. Dicha alternativa segura se presentará en el Capítulo 5.

3.3.4 Borrado

La operación de borrado sigue una estructura recursiva similar a las de inserción y búsqueda, pero distingue algunos casos particulares adicionales.

La estrategia es reemplazar el nodo que se borra con el nodo anterior en el orden del árbol, que resulta ser el nodo con clave máxima del subárbol izquierdo. Al igual que con la inserción, se aplica `balance` luego de cada llamada recursiva a `delete`. Para hallar el nodo con clave máxima del subárbol izquierdo se utiliza la función auxiliar `maxNode`. Se elimina el nodo con clave máxima con la función auxiliar `maxKeyDelete`.

```
deleteAVL :: Int → AVL a → AVL a
deleteAVL _ E                    = E
deleteAVL x t@(F _ (Node n _) _) = deleteAVL' x t (compare x n)

deleteAVL' :: Int → AVL a → Ordering → AVL a
deleteAVL' _ (F E     _ E)     EQ = E
deleteAVL' _ (F E     _ r@F{}) EQ = r
deleteAVL' _ (F l@F{} _ E)     EQ = l
deleteAVL' _ (F l@F{} _ r@F{}) EQ =
  balance $ FF (maxKeyDelete l) mNode r
    where Just mNode = maxNode l
deleteAVL' _ t@(F E _ _) LT = t
deleteAVL' x (F l@(F _ (Node ln _) _) node r) LT =
  balance $ FF (deleteAVL' x l (compare x ln)) node r
deleteAVL' _ t@(F _ _ E) GT = t
deleteAVL' x (F l node r@(F _ (Node rn _) _)) GT =
  balance $ FF l node (deleteAVL' x r (compare x rn))
```

Código 3.11: Borrado no seguro.

Luego de borrar un nodo, es necesario probar que el nuevo árbol mantiene los invariantes de orden en las claves y alturas balanceadas:

Teorema 3.3.6. Si `x::Int` y `t` es *BST*, entonces `delete x t` también es *BST*.

Teorema 3.3.7. Si `x::Int` y `t` es un árbol *balanceado*, entonces `delete x t` también es un árbol *balanceado*.

Como `delete` utiliza la función auxiliar `maxKeyDelete`, la demostración de los enunciados anteriores puede valerse de los lemas siguientes:

Lema 3.3.2. Si `t` es *BST*, entonces `maxKeyDelete t` también es *BST*.

Lema 3.3.3. Si `t` es un árbol *balanceado*, entonces `maxKeyDelete t` es un árbol *casi balanceado*.

3.4 Desempeño

Se evaluó el desempeño de esta implementación y se lo comparó con una implementación de árboles `BST` implementada de manera similar (es decir, sin programación a nivel de tipos).

La medida de desempeño elegida es el tiempo de ejecución. Se midió el tiempo de insertar/borrar/buscar una clave en árboles con N cantidad de nodos, para distintos valores de N. Los árboles fueron construidos insertando nodos con valores de claves consecutivas, 0, 1, 2, etc., de modo que los árboles `BST` tienen la estructura de una lista. Allí se insertó/borró/buscó el nodo con clave máxima, es decir el último nodo de la lista. Este es uno de los peores casos para las operaciones. Por más detalles, consultar el apéndice A.

En ambas implementaciones, los tiempos de inserción y búsqueda muestran una tendencia lineal. Los árboles `BST` muestran un mejor desempeño, ya que las operaciones sobre árboles `AVL` incorporan el cálculo de alturas.

En los árboles *AVL* de [AL62], estas operaciones tienen implementaciones de orden $O(\log N)$. Esta implementación difiere de la citada en que no se mantiene información extra en los nodos sobre las alturas de los sub árboles; en su lugar, se calcula el valor de las alturas cada vez que sea necesario. Es por esa razón que la inserción y el borrado muestran tiempos de ejecución de orden $O(N)$.

En cambio, la búsqueda en árboles `AVL` muestra un mejor desempeño que en árboles `BST`.

No fue posible mostrar que el tiempo de ejecución de la búsqueda en árboles `AVL` se ajusta a $O(\log N)$, porque los valores obtenidos son similares entre sí y solo muestran una tendencia constante (se encuentran en el rango de 2.54×10^{-6} a 7.16×10^{-6}). La mayor cantidad de nodos que fue posible utilizar con los recursos computacionales disponibles fue de 2^{15}, por lo que los árboles `AVL` utilizados tienen como máximo una altura de 15. Es por esto que los tiempos de búsqueda no son significativamente diferentes.

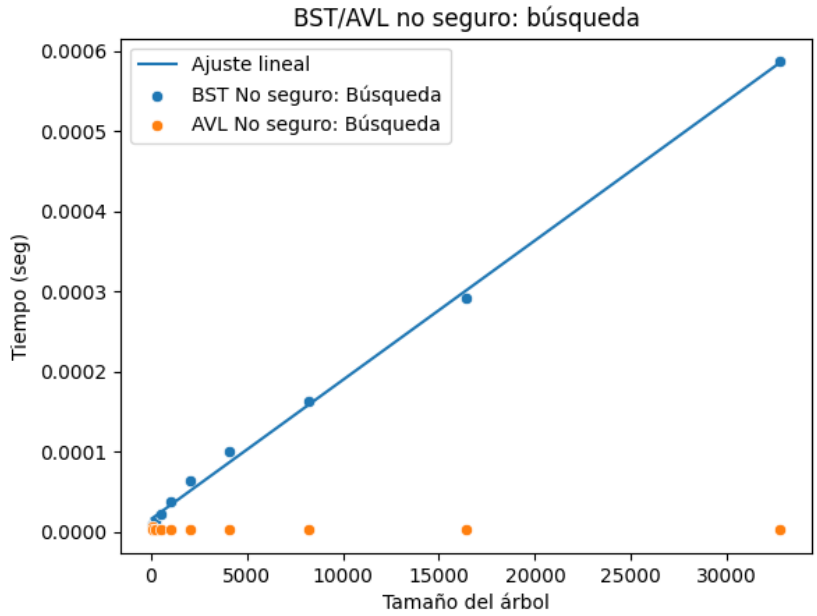


Figura 3.4: Tiempos de ejecución de la búsqueda con el enfoque no seguro en árboles `BST` y `AVL` de distintos tamaños.

En el Apéndice D se encuentran todos los valores de tiempos y en el Apéndice C las gráficas de estos. Por más detalles sobre el ajuste lineal utilizado y la ejecución de los benchmarks, ver los Apéndices A y B.

Capítulo 4

Implementación segura de árboles BST

En este capítulo se implementan los árboles *BST* como un *GADT* con restricciones en los constructores, de forma tal que se verifica el orden de las claves en tiempo de compilación. El lugar en donde se coloca las restricciones depende del enfoque utilizado (externalista vs internalista).

Los árboles `BST` implementados almacenan elementos que contienen una clave y un valor. La clave es un número natural a nivel de tipos, es decir, la clave existe solamente en el tipo del árbol. Al igual que en la implementación no segura, los valores pueden ser de cualquier tipo (siempre de *kind* `Type`).

Esta clase de colecciones, capaces de contener datos de tipos diferentes, se denominan colecciones heterogéneas. Otro ejemplo de colección heterogénea son las listas heterogéneas [KLS04]. Tienen una implementación en la librería `HList`[KLS18], que va más allá de una simple colección heterogénea, incorporando, a nivel de tipos, el largo de la lista y el tipo de cada uno de sus elementos. Esto permite garantizar ciertas propiedades sobre la estructura de datos o sus operaciones (e.g., que la lista contiene al menos un elemento, que la búsqueda reporte un resultado, etc). Esta implementación de árboles *BST*, y la de árboles *AVL* que se presenta en el capítulo siguiente, siguen este enfoque.

4.1 Constructores

Los nodos de los árboles se definen como un *GADT*, donde la clave es un *phantom type* de *kind* `Nat`. Un *phantom type* es un tipo que no tiene un valor asociado en el constructor [CH03].

```
data Node :: Nat → Type → Type where
  Node :: a → Node k a
```

Código 4.1: Constructor de nodos.

A nivel de valores, los nodos almacenan un valor de cualquier tipo (de *kind* `Type`); a nivel de tipos, contienen la clave y el tipo del valor almacenado. Se puede definir la clave de un nodo mediante una signatura de tipo

```
-- node1 :: Node 4 Char, la clave es 4.
node = Node 'a' :: Node 4 Char
```

Código 4.2: Ejemplo de nodo.

o utilizando una función que recibe la clave como parámetro a través de un *proxy*

```
mkNode :: Proxy k → a → Node k a
mkNode _ = Node

node :: Node 4 Char
node = mkNode (Proxy::Proxy 4) 'a'
```

Código 4.3: Constructor de nodos. Ejemplo de nodo.

Por otro lado, se maneja una estructura de datos sobre la que se van a definir los invariantes de árbol *BST*:

```
data Tree :: Type where
  EmptyTree :: Tree
  ForkTree  :: Tree → n → Tree → Tree
```

Código 4.4: Constructor de árboles de tipo `Tree`.

En ambos enfoques (externalista e internalista), los árboles `BST` son un *GADT* que está indexado por árboles a nivel de tipos de *kind* `'Tree` (tipos promovidos desde el *GADT* `Tree` por la extensión *DataKinds*):

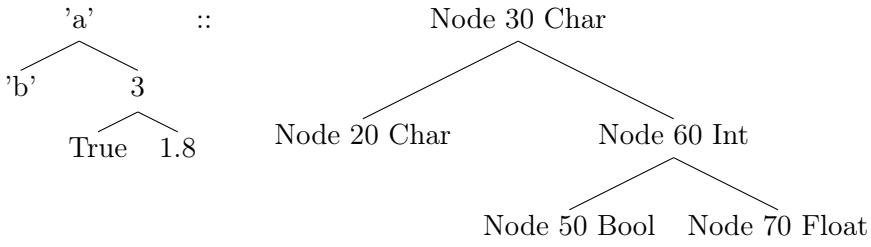
```
data BST :: Tree → Type where
  ...
```

Código 4.5: El tipo de los árboles `BST` está indexado por un árbol de *kind* `Tree`.

Es decir, los árboles `BST` tienen tipo `BST t`, donde `t` es un tipo de *kind* `Tree` y es la representación a nivel de tipos del árbol (ambos árboles tienen la misma estructura).

Mediante técnicas de programación a nivel de tipos, es posible verificar en tiempo de compilación para cada árbol de tipo `BST t` si las claves en el árbol a nivel de tipos `t` están ordenadas. Luego, como ambos árboles son isomorfos en su estructura, se puede asegurar que los árboles a nivel de valores son *BSTs*.

Por ejemplo:



La verificación de que las claves $20, 30, 50, 60, 70$ se encuentran ordenadas es realizada en tiempo de compilación.

4.2 Invariantes

Se verifica el orden en las claves a nivel de tipos (se considera el orden usual de los números naturales). No se admiten claves repetidas. Se implementa mediante *type families*.

Para ordenar las claves, se debe asegurar que en cada nodo se cumple que las claves del lado izquierdo son menores y las del lado derecho mayores que la del nodo.

Se define, de forma recursiva, una *type family* para decidir si todas las claves en los nodos de un árbol son menores que una clave particular:

```
type family LtN (t::Tree) (x::Nat) :: Bool where
  LtN 'EmptyTree                  x = 'True
  LtN ('ForkTree l (Node n a) r) x =
    CmpNat n x == 'LT && LtN l x && LtN r x
```

Código 4.6: Verificar si las claves de `t` son menores que `x`.

Esta *type family* se puede considerar como una función (a nivel de tipos) $LtN :: Tree \to Nat \to Bool$, que dado un árbol y una clave, retorna `'True` si y solo si todas las claves del árbol son menores que la clave dada. `CmpNat` es otra *type family*, definida en `GHC.TypeLits`, que para cada par de naturales retorna el resultado de su comparación como uno de tres valores posibles: `'LT`, `'EQ` o `'GT`.

Análogamente, se define una type family `GtN`, para decidir si todas las claves de un árbol son mayores que una clave dada:

```
type family GtN (t::Tree) (x::Nat) :: Bool where
  GtN 'EmptyTree                  x = 'True
  GtN ('ForkTree l (Node n a) r) x =
    CmpNat n x == 'GT && GtN l x && GtN r x
```

Código 4.7: Verificar si las claves de `t` son mayores que `x`.

4.3 Enfoque completamente externalista

Este enfoque consiste en operar directamente sobre valores de tipo `ITree`, un tipo *singleton* indexado en árboles de *kind* `Tree`, el cual no tiene ninguna verificación de invariantes de *BST*. Se aplica el constructor de `BST` una sola vez, al final, luego de haber aplicado todas las operaciones sobre el árbol `ITree`, para verificar el orden en las claves.

4.3.1 Constructores

```
data ITree :: Tree → Type where
```

```
  EmptyITree :: ITree 'EmptyTree
  ForkITree  :: Show a => ITree l → Node n a → ITree r
    → ITree ('ForkTree l (Node n a) r)
```

Código 4.8: Constructor de árboles singleton.

En los enfoques externalistas, la evidencia del cumplimiento de los invariantes está desacoplada de los valores y tiene una representación a nivel de valores, un "término de prueba":

```
data BST :: Tree → Type where
  BST :: ITree t → IsBSTT t → BST t
```

Código 4.9: Constructor externalista de árboles `BST`.

El valor de tipo `ITree t` es el árbol *BST*, mientras que el valor de tipo `IsBSTT t` representa el término de prueba que contiene, a nivel de tipos, la información necesaria para demostrar que las claves de `t` se encuentran ordenadas:

```
data IsBSTT :: Tree → Type where
  EmptyIsBSTT :: IsBSTT 'EmptyTree
  ForkIsBSTT  :: (LtN l n ~ 'True, GtN r n ~ 'True) =>
    IsBSTT l → Proxy (Node n a) → IsBSTT r →
    IsBSTT ('ForkTree l (Node n a) r)
```

Código 4.10: Término de prueba para árboles `BST`.

La estructura del término de prueba es isomorfa a la del árbol. A nivel de valores sin embargo, los nodos del término de prueba no son utilizados para almacenar ningún valor relevante. Esto queda evidenciado por su tipo, `Proxy (Node n a)`. El rol de esta estructura es solamente proveer evidencia del orden en las claves.

Si bien la evidencia del orden de las claves tiene su representación a nivel de valores por un término de tipo `IsBSTT`, la verificación del orden se realiza en realidad a nivel de tipos, en tiempo de compilación. El constructor `ForkIsBSTT` tiene las restricciones `LtN l n ∼ 'True`, `GtN r n ∼ 'True`, que se definen a nivel de tipos y son las que verifican el orden en las claves. Estas restricciones son verificadas al momento de aplicar el constructor. Luego de construido el árbol, al realizar *pattern matching* sobre el constructor, se incorpora al contexto las igualdades de tipos, de forma análoga al constructor `Refl` de la Sección 2.9.2.

En particular, para el caso completamente externalista, se define un constructor `mkBST`, que hace uso de una *type class* `IsBSTC` para construir el término de prueba automáticamente, gracias al mecanismo de inferencia de instancias de *type classes*:

```
class IsBSTC (t :: Tree) where
  isBSTT :: IsBSTT t

instance IsBSTC 'EmptyTree where
  isBSTT = EmptyIsBSTT
```

```
instance (IsBSTC l, IsBSTC r, LtN l n ~ 'True, GtN r n ~ 'True) =>
  IsBSTC ('ForkTree l (Node n a) r) where
  isBSTT = ForkIsBSTT isBSTT (Proxy :: Proxy (Node n a)) isBSTT
```

Código 4.11: *Type class* para la construcción automática del término de prueba `IsBSTT`.

En la segunda instancia de `IsBSTC`, el nodo raíz se define como `Proxy::Proxy (Node n a)`; es decir el término de prueba no contiene información a nivel de valores.

Por otro lado, en el contexto de la misma instancia se encuentran cuatro restricciones: `IsBSTC l`, `IsBSTC r`, `LtN l n ~ 'True` y `GtN r n ~ 'True`. Las dos primeras requieren de los términos de prueba para los sub árboles izquierdo y derecho. Esto desencadena el proceso automático de contrucción del término de prueba mediante el mecanismo de inferencia de instancias. Las otras dos restricciones verifican el orden de las claves y son necesarias para poder aplicar el constructor `ForkIsBSTT`.

```
mkBST :: (IsBSTC t) => ITree t → BST t
mkBST t = BST t isBSTT
```

Código 4.12: Constructor `mkBST`.

En el contexto de `mkBST` se pide la instancia de la clase `IsBSTC` para `t`, la cual construye automáticamente el término de prueba para `t`, de tipo `IsBSTT t`, en la llamada al método `isBSTT`. De este modo, el usuario puede operar libremente en el árbol de tipo `ITree t` y, cuando finalmente desee verificar si es *BST*, simplemente aplica el constructor `mkBST`.

Si el `ITree` al que se aplica `mkBST` no es *BST*, entonces la compilación falla porque no se encuentran las instancias de `IsBSTC`. Ejemplos de aplicación y casos de error se muestran con más detalle en la Sección 4.3.5.

4.3.2 Inserción

Como las claves solo existen a nivel de tipos, el comportamiento de la función de inserción (y de cualquier función que manipule o consulte las claves) necesariamente depende de la información a nivel de tipos. Por ejemplo, al insertar un elemento en un árbol de tipo `ITree t`, es necesario inspeccionar el tipo `t` para hallar el lugar en donde insertar el elemento, además de modificar `t`. Es decir, la implementación depende directamente del tipo del árbol. De modo que `insert` es necesariamente polimórfica ad-hoc.

Se definen dos clases. La clase principal `Insertable`

```
class Insertable (x::Nat) (a::Type) (t::Tree) where
  type Insert (x::Nat) (a::Type) (t::Tree) :: Tree
  insert :: Node x a → ITree t → ITree (Insert x a t)
```

Código 4.13: `Insertable` completamente externalista.

y una clase auxiliar `Insertable'` que es invocada por la primera

```
class Insertable' (x::Nat) (a::Type) (t::Tree) (o::Ordering) where
  type Insert' (x::Nat) (a::Type) (t::Tree) (o::Ordering) :: Tree
  insert' :: Node x a → ITree t → Proxy o → ITree (Insert' x a t o)
```

Código 4.14: `Insertable'` completamente externalista.

`Insertable` es la interfaz para invocar la operación de inserción, mientras que `Insertable'` es en donde se encuentra la lógica de la operación. El parámetro `o::Ordering` guía el proceso de inserción y corresponde a la comparación entre la nueva clave y la clave del nodo raíz.

Cada caso posible de inserción queda definido en una instancia distinta de la clase. Por ejemplo

```
instance (l ~ 'ForkTree ll (Node ln lna) lr, o ~ CmpNat x ln, Insertable' x a l o) ⇒
  Insertable' x a ('ForkTree ('ForkTree ll (Node ln lna) lr) (Node n a1) r) 'LT where
  type Insert' x a ('ForkTree ('ForkTree ll (Node ln lna) lr) (Node n a1) r) 'LT =
    'ForkTree (Insert' x a ('ForkTree ll (Node ln lna) lr) (CmpNat x ln)) (Node n a1) r
  insert' node (ForkITree l node' r) _ =
    ForkITree (insert' node l (Proxy::Proxy o)) node' r
```

Código 4.15: Ejemplo de instancia de la clase `Insertable'`.

Para ello se hace uso de la extensión *Scoped Type Variables*. La variable de tipo `o` que aparece en el contexto de la instancia y en `Proxy::Proxy o` es la misma.

En esta instancia, el parámetro `o::Ordering` es `'LT`, lo que indica que la nueva clave es menor que la clave de la raíz. Luego, se invoca a `insert'` recursivamente en la rama izquierda. La implementación de la aplicación recursiva de `insert'` corresponde a la instancia `Insertable' x a l o`, definida como una *class constraint*.

4.3.3 Búsqueda

El algoritmo de búsqueda no modifica el árbol, aun así el diseño es similar a `insert`: hay una clase principal, `Lookupable`, la interfaz para el algoritmo de búsqueda, que a su vez depende de otra clase, `Lookupable'`, que agrega un parámetro `o::Ordering` para guiar el proceso de búsqueda.

```
class Lookupable (x::Nat) (a::Type) (t::Tree) where
  lookup :: (t ~ 'ForkTree l (Noden a1) r, Member x t ~ 'True) ⇒
    Proxy x → ITree t → a
```

Código 4.16: `Lookupable` completamente externalista.

```
class Lookupable' (x::Nat) (a::Type) (t::Tree) (o::Ordering) where
  lookup' :: Proxy x → ITree t → Proxy o → a
```

Código 4.17: `Lookupable'` completamente externalista.

En la implementación de esta operación se muestra una de las ventajas de las técnicas de programación a nivel de tipos. En el contexto de la función `lookup`, aparecen las igualdades de tipos `t ∼ 'ForkTree l (Node n a1) r` y `Member x t ∼ 'True`, donde `Member` es una *type family* que evalúa `'True` si y solo si la clave `x` se encuentra en el árbol `t`:

```
type family Member (x::Nat) (t::Tree) :: Bool where
  Member x 'EmptyTree = 'False
  Member x ('ForkTree l (Node n a) r) =
    (If (CmpNat x n == 'EQ)
      'True
      (If (CmpNat x n == 'LT)
        (Member x l)
        (Member x r)
      )
    )
```

Código 4.18: Evaluar a nivel de tipos si una clave se encuentra en el árbol.

De este modo, en tiempo de compilación se determina si la clave se encuentra en el árbol.

Con el enfoque no seguro, se hizo uso del tipo `Maybe` para manejar el caso en que la clave buscada no se encuentra en el árbol. Una alternativa podría ser retornar un valor especial o lanzar una excepción. En cambio, en el enfoque seguro, las restricciones a nivel de tipos capturan este caso en tiempo de compilación.

4.3.4 Borrado

El diseño e implementación del borrado es similar al de la inserción. La diferencia se encuentra en los casos recursivos adicionales, como se mostró en la Sección 3.3.4.

4.3.5 Mensajes de error

La principal ventaja del enfoque completamente externalista es que la verificación del orden de las claves se realiza solamente una vez, después de realizar todas las operaciones necesarias sobre el árbol. Por ejemplo, se insertan varios elementos y al final se aplica el constructor `BST` para verificar si el árbol es *BST*:

```
bst = mkBST t
  where
    t = insert (mkNode p4 'f')
      $ insert (mkNode p2 4)
      $ insert (mkNode p6 "la")
      $ insert (mkNode p3 True)
      $ insert (mkNode p5 [1,2,3])
      $ insert (mkNode p0 1.8)
      $ insert (mkNode p7 [False]) emptyTree
```

Código 4.19: Ejemplo de construcción de árbol BST en el enfoque completamente externalista.

donde `p2`, `p3`, `p4`, etc., son *proxies* con un valor natural a nivel de tipos (2, 3, 4, etc., respectivamente).

En el ejemplo se insertan elementos sucesivamente, comenzando con el árbol vacío. La operación `insert` actúa sobre árboles `ITree`, los cuales no tienen ninguna restricción en el orden de sus claves. Al aplicar el constructor `mkBST`, el compilador construye la instancia `IsBSTC t`, que contiene el término de prueba `isBSTT::IsBSTT t`. En otras palabras, si la compilación es exitosa, el compilador verificó que se puede construir la evidencia de que el árbol es *BST*.

Un caso de error en la compilación es el siguiente:

```
bstError = mkBST $
  ForkITree (ForkITree EmptyITree (mkNode p7 4) EmptyITree)
            (mkNode p4 'f')
            EmptyITree
```

Código 4.20: Caso de error en la aplicación del constructor `mkBST`.

Aquí no se cumple que las claves estén ordenadas (un nodo con clave 7 se encuentra a la izquierda de uno con clave 4). Se obtiene el siguiente error de compilación:

```
• Couldn't match type ''False' with ''True'
arising from a use of 'mkBST'
• In the expression:
mkBST
$ ForkITree
(ForkITree EmptyITree (mkNode p7 4) EmptyITree) (mkNode p4 'f')
EmptyITree
...
```

Como el árbol no es *BST*, no se puede construir la instancia correspondiente de `IsBSTC`. Si `n` es la raíz y `l` es el subárbol izquierdo, se tiene que `LtN l n` $\sim$ `'False`, ya que la clave 7 de `l` no es menor que la clave 4 de `n`. Pero la restricción de la instancia de `IsBSTC` exige `LtN l n` $\sim$ `'True`. Luego, el compilador intenta resolver `'False` $\sim$ `'True`. De ahí el mensaje "`Couldn't match type ''False' with ''True'`". Por otro lado, el mensaje de error también indica en que parte del código se encuentra el mismo: "`arising from a use of 'mkBST' In the expression:...`".

Sin embargo, el mensaje no indica explícitamente que el error se encontró en la restricción de la instancia de `IsBSTC`. Un mensaje de error más informativo agregaría que el error se produjo en la igualdad de tipos `LtN l n` $\sim$ `'True`, informando además las instancias de `l` y `n`.

En este caso se presenta la necesidad de implementar mensajes de error a medida:

```
type family LtN (l :: Tree) (x :: Nat) :: Bool where
  LtN 'EmptyTree                  x = 'True
  LtN ('ForkTree l (Node n a) r) x =
    (If (CmpNat n x == 'LT)
```

```
      (If (LtN l x)
        (If (LtN r x)
          'True
          (TypeError (Text "Key " :<>: ShowType x :<>:
            Text " is not smaller than keys in tree " :<>: ShowType r))
        )
        (TypeError (Text "Key " :<>: ShowType x :<>:
          Text " is not smaller than keys in tree " :<>: ShowType l))
      )
      (TypeError (Text "Key " :<>: ShowType x :<>: Text " is not smaller than key " :<>:
        ShowType n))
    )
```

Código 4.21: *Type family* `LtN` con mensajes de error específicos.

```
type family GtN (r :: Tree) (x :: Nat) :: Bool where
  GtN 'EmptyTree                  x = 'True
  GtN ('ForkTree l (Node n a) r) x =
    (If (CmpNat n x == 'GT)
      (If (GtN l x)
        (If (GtN r x)
          'True
          (TypeError (Text "Key " :<>: ShowType x :<>:
            Text " is not greater than keys in tree " :<>: ShowType r))
        )
        (TypeError (Text "Key " :<>: ShowType x :<>:
          Text " is not greater than keys in tree " :<>: ShowType l))
      )
      (TypeError (Text "Key " :<>: ShowType x :<>: Text " is not greater than key " :<>:
        ShowType n))
    )
```

Código 4.22: *Type family* `GtN` con mensajes de error específicos.

Con estas modificaciones (utilizando las herramientas presentadas previamente en la Sección 2.6), el nuevo mensaje de error para el ejemplo anterior pasa a ser

```
• Key 7 is not smaller than key 4
• In the first argument of ‘($)’, namely ‘mkBST’
In the expression:
mkBST
$ ForkITree
(ForkITree EmptyITree (mkNode p7 4) EmptyITree) (mkNode p4 'f')
EmptyITree
...
```

Con este mensaje queda evidenciada la raíz del problema: los nodos de claves 4 y 7 no se encuentran ordenados apropiadamente.

Otro ejemplo de error se puede tener en el uso de `lookup`:

```
-- | Error: key 1 is not in the tree bst
err = case bst of
```

```
  BST t _ → lookup p1 t
```

Código 4.23: Caso de error: la clave buscada no se encuentra en el árbol.

donde `bst` es el árbol del primer ejemplo.

La operación de `lookup` tiene la restricción `Member x t ∼ 'True`. En este caso, como ningún nodo de `bst` tiene clave 1, la igualdad de tipos no se cumple. Al igual que antes, el mensaje de error indica la ubicación en el código en dónde se produjo el mismo, pero no exactamente cual es la igualdad de tipos que no se pudo verificar:

```
• Couldn't match type ''False' with ''True'
arising from a use of 'lookup'
• In the expression: lookup p1 t
In a case alternative: BST t _ -> lookup p1 t
In the expression: case bst of { BST t _ -> lookup p1 t }
```

Luego de modificar la implementación de la *type family* `Member`, incorporando un mensaje de error más apropiado

```
type family Member (x :: Nat) (t :: Tree) (t' :: Tree) :: Bool where
  Member x ('ForkTree l (Node n a) r) t' =
    (If (CmpNat x n == 'EQ)
      'True
      (If (CmpNat x n == 'LT)
        (Member x l t')
        (Member x r t')
      )
    )
  Member x 'EmptyTree t' = TypeError (Text "Key " :<>: ShowType x :<>:
    Text " not found in tree " :<>: ShowType t')
```

Código 4.24: *Type family* `Member` con mensajes de error específicos.

se obtiene un nuevo mensaje de error (acortado por razones de espacio) mucho más informativo

```
• Key 1 not found in tree 'Data.Tree.ITree.ForkTree
...
• When checking the inferred type
err :: (TypeError ...)
```

donde se muestra la clave que no se pudo encontrar y el árbol en donde la búsqueda fue realizada.

El tercer parámetro de la *type family* `Member` propaga el árbol sobre el cual la *type family* fue invocada en primera instancia, a modo de contexto para la construcción del mensaje de error.

4.4 Enfoque externalista

En el enfoque externalista, cada vez que se realiza una operación sobre un árbol `BST` se extrae el árbol `ITree`, se realiza la operación y se aplica el constructor de

`BST` al resultado. Si el árbol es modificado, es necesario actualizar el término de prueba.

Se utilizan los mismos constructores `ITree` y `BST` que en el enfoque presentado anteriormente. La diferencia es que ahora se implementan las operaciones sobre árboles de tipo `BST t` en lugar de `ITree t`.

```
insertBST :: (Insertable x a t, ProofIsBSTInsert x a t) =>
  Proxy x → a → BST t → BST (Insert x a t)
insertBST (px :: Proxy x) (a :: a) (BST t tIsBST) =
  BST (insert node t) (proofIsBSTInsert pNode tIsBST)
    where node  = mkNode px a
          pNode = Proxy :: Proxy (Node x a)
```

Código 4.25: Inserción con el enfoque externalista.

Se reutiliza la implementación de `insert` de la *type class* `Insertable` del enfoque anterior.

La anotación de tipo en los primeros dos argumentos, `(px::Proxy x)` y `(a::a)`, es necesaria. En principio, el alcance de las variables de tipo en la firma de tipo de `insertBST` no va más allá de la propia firma, por lo que no alcanza a la definición de la función. En ese caso, las variables de tipo `x` y `a` en el valor `pNode = Proxy::Proxy (Node x a)` son variables nuevas, diferentes a las que aparecen en la firma de tipo de `insertBST`. Esto genera un error de compilación. Para unificar estas variables, se utiliza la extensión *Scoped Type Variables*. La extensión, en conjunción con la signatura de tipo en los argumentos[20o], permite extender el alcance de las variables de la signatura de tipo de `insertBST` al cuerpo de su definición.

`proofIsBSTInsert` proviene de la instancia `ProofIsBSTInsert x a t`. Construye el término de prueba que muestra que insertar la clave `x` y un elemento de tipo `a` en un árbol *BST* `t` retorna un árbol que también es *BST*.

```
class ProofIsBSTInsert (x::Nat) (a::Type) (t::Tree) where
  proofIsBSTInsert :: Proxy (Node x a) → IsBSTT t → IsBSTT (Insert x a t)
```

Código 4.26: *Type class* `ProofIsBSTInsert`. Demuestra que `Insert'` preserva el orden de las claves.

Esta función es la implementación del Teorema 3.3.4 que establece la preservación del orden de las claves para la inserción. De su tipo se deduce que su función es construir el término de prueba para el árbol que resulta de la inserción partiendo del término de prueba original.

Más adelante se muestra la implementación de estas funciones de prueba, utilizadas para ayudar al compilador en la inferencia de igualdades de tipos, y las implicancias que tienen en el desarrollo de código correcto en cuanto al cumplimiento de los invariantes en los árboles.

La búsqueda también se implementa reutilizando su contraparte del enfoque completamente externalista:

```
lookupBST :: (t ~ 'ForkTree l (Node n a1) r, Member x t t ~ 'True,
    Lookupable x a t) =>
  Proxy x -> BST t -> a
  lookupBST p (BST t _) = lookup p t
```

Código 4.27: Búsqueda con el enfoque externalista.

La implementación del borrado es análoga al de la inserción:

```
deleteBST :: (Deletable x t, ProofIsBSTDelete x t) =>
  Proxy x -> BST t -> BST (Delete x t)
  deleteBST px (BST t tIsBST) = BST (delete px t) (proofIsBSTDelete
    px tIsBST)
```

Código 4.28: Borrado con el enfoque externalista.

Nuevamente, se utiliza una función de prueba `proofIsBSTDelete` para construir el término de prueba para el nuevo árbol. Esta función implementa el Teorema 3.3.6 sobre la preservación del orden de las claves bajo el borrado.

4.5 Enfoque internalista

En el enfoque completamente externalista la verificación del invariante de `BST` se realiza una sola vez, luego de aplicar una serie de operaciones sobre un árbol `ITree`. En el externalista la verificación se realiza luego de cada operación que modifique el árbol. Sin embargo, en el enfoque internalista la verificación y la evidencia se acercan aún más al proceso de construcción del árbol.

4.5.1 Constructores

A diferencia del enfoque externalista, el internalista no maneja un árbol `ITree`. Los invariantes son verificados cada vez que se agrega un nuevo nodo al árbol:

```
data BST :: Tree -> Type where
  EmptyBST :: BST 'EmptyTree
  ForkBST  :: (Show a, LtN l n ~ 'True, GtN r n ~ 'True) =>
    BST l -> Node n a -> BST r -> BST ('ForkTree l (Node n a) r)
```

Código 4.29: Constructor internalista de árboles `BST`.

Es posible demostrar mediante inducción que los árboles construidos con `EmptyBST` y `ForkBST` son árboles *BST* (tienen sus claves ordenadas). En el caso base, el árbol vacío es *BST*. En el caso inductivo, asumiendo que `l` y `r` son *BST*, para cualquier nodo `node::Node n a` con clave `n`, el árbol `ForkBST l node r` es *BST* porque:

- `l` es *BST*,
- `r` es *BST* y
- las restricciones `LtN l n ~ 'True`, `GtN r n ~ 'True` del constructor `ForkBST` verifican (si el programa compila correctamente) que todas las claves de `l` son menores que la clave de la raíz y que las de `r` son mayores.

Al haber modificado el constructor de árboles `BST`, es necesario reimplementar las operaciones de inserción, búsqueda y borrado.

4.5.2 Operaciones

La implementación de la inserción es similar a su contraparte del enfoque externalista, con la diferencia de que ahora la función `insert` opera sobre árboles de tipo `BST` en lugar de `ITree`:

```
class Insertable (x::Nat) (a::Type) (t::Tree) where
  type Insert (x::Nat) (a::Type) (t::Tree) :: Tree
  insert :: Node x a → BST t → BST (Insert x a t)
```

Código 4.30: `Insertable` en el enfoque internalista. Se inserta directamente en un `BST`, en lugar de un `ITree`.

Tampoco se invoca ninguna función de prueba luego de realizar la operación, sino que las demostraciones necesarias son aplicadas dentro de la lógica de las operaciones. Por ejemplo, en la siguiente instancia de `Insertable'`

```
instance (l ~ 'ForkTree ll (Node ln lna) lr, o ~ CmpNat x ln,
  CmpNat x n ~ 'LT,
  Insertable' x a l o,
  ProofLtNInsert' x a ('ForkTree ll (Node ln lna) lr) n o) ⇒
  Insertable' x a ('ForkTree ('ForkTree ll (Node ln lna) lr) (Node n a1) r) 'LT where
    type Insert' x a ('ForkTree ('ForkTree ll (Node ln lna) lr) (Node n a1) r) 'LT =
      'ForkTree (Insert' x a ('ForkTree ll (Node ln lna) lr) (CmpNat x ln)) (Node n a1) r
    insert' node (ForkBST l node' r) _ =
      gcastWith (proofLtNInsert' node l (Proxy::Proxy n) po) $
      ForkBST (insert' node l po) node' r
        where
          po = Proxy::Proxy (CmpNat x ln)
```

Código 4.31: Ejemplo de instancia para la inserción internalista.

se aplica la función de prueba `proofLtNInsert'` luego de construir el árbol que resulta de la inserción para este caso.

Al realizar *pattern matching* sobre el árbol que se recibe como argumento en `ForkBST l node' r`, se incorporan al contexto las restricciones `LtN l n ~ 'True` y `GtN r n ~ 'True`, definidas en el constructor `ForkBST`.

Para construir el árbol `ForkBST (insert' node l po) node' r`, se deben cumplir las igualdades de tipos `LtN (Insert' x a l o) n ~ 'True` y `GtN r n ~ 'True`. La función de prueba `proofLtNInsert'` se encarga de construir la evidencia de la primera igualdad a partir de que `LtN l n ~ 'True`, mientras que la segunda viene dada por el árbol original.

La implementación de la búsqueda es similar a los casos externalistas. Se realiza una verificación a nivel de tipos con la *type family* `Member` para saber si la clave buscada se encuentra en el árbol.

```
lookupBST :: (t ~ 'ForkTree l (Node n a1) r, Member x t t ~ 'True,
    Lookupable x a t) ⇒
```

```
  Proxy x → BST t → a
lookupBST = lookup
```

Código 4.32: Búsqueda en el enfoque internalista.

Las diferencias en la implementación del borrado entre los enfoques externalista e internalista son similares a las de la inserción.

4.6 Pruebas

En general las pruebas principales son sobre la verificación del invariante de orden en las claves y refieren a una operación en particular. Es decir, se implementa un conjunto de funciones de prueba para demostrar que la inserción mantiene el orden en las claves y otro conjunto para demostrar que el borrado mantiene el orden. Estas funciones de prueba son la implementación de los teoremas que se presentaron en el Capítulo 3.

Al igual que ocurre con las operaciones de inserción, búsqueda y borrado, la implementación de las funciones de prueba depende de los valores de las claves, que se encuentran a nivel de tipos, por lo que se implementan mediante *type classes*. Se define una única función de prueba por cada *type class*, para seleccionar solamente las funciones de prueba que se necesitan a través de la instancia de *type class* correspondiente.

La implementación de las pruebas es similar en ambos enfoques externalista e internalista. A continuación se presenta el diseño de las pruebas para el primer enfoque y luego se realiza una comparación con respecto al segundo.

Por ejemplo, para la inserción, se recuerda que el teorema correspondiente es

Teorema. Si `x::Int`, `a::a` y `t` es *BST*, entonces `insert x a t` también es *BST*.

La *type class* que implementa este teorema es

```
class ProofIsBSTInsert (x::Nat) (a::Type) (t::Tree) where
  proofIsBSTInsert :: Proxy (Node x a) → IsBSTT t → IsBSTT (Insert x a t)
```

Código 4.33: *Type class* `ProofIsBSTInsert`. Implementación del Teorema 3.3.4.

La función de prueba recibe como argumentos un *proxy* del nodo que se va a insertar y el término de prueba del árbol original, y construye el término de prueba para el árbol que resulta de insertar el nodo. La función de prueba realiza la misma inserción que la función `insert`, pero en el término de prueba, y el nodo que se inserta no contiene ninguna información a nivel de valores, es un *proxy* para la información a nivel de tipos del nodo.

`ProofIsBSTInsert` depende de otra *type class*, `ProofIsBSTInsert'`, que agrega un parámetro `o::Ordering` para guiar la inserción en el término de prueba

```
class ProofIsBSTInsert' (x::Nat) (a::Type) (t::Tree) (o::Ordering) where
  proofIsBSTInsert' :: Proxy (Node x a) → IsBSTT t → Proxy o →
    IsBSTT (Insert' x a t o)
```

Código 4.34: *Type class* `ProofIsBSTInsert'`. Demuestra que `Insert'` preserva el orden de las claves.

A su vez, esta *type class* utiliza otras dos *type classes*: `ProofLtNInsert'` y `ProofGtNInsert'`:

```
class ProofLtNInsert' (x::Nat) (a::Type) (t::Tree) (n::Nat) (o::Ordering) where
  proofLtNInsert' :: (CmpNat x n ~ 'LT, LtN t n ~ 'True) =>
    Proxy (Node x a) → IsBSTT t → Proxy n → Proxy o →
      LtN (Insert' x a t o) n :~: 'True
```

Código 4.35: *Type class* `ProofLtNInsert'`. Demuestra que `Insert'` preserva el invariante `LtN`. Implementación del Lema 3.3.1.

```
class ProofGtNInsert' (x::Nat) (a::Type) (t::Tree) (n::Nat) (o::Ordering) where
  proofGtNInsert' :: (CmpNat x n ~ 'GT, GtN t n ~ 'True) =>
    Proxy (Node x a) → IsBSTT t → Proxy n → Proxy o →
      GtN (Insert' x a t o) n :~: 'True
```

Código 4.36: *Type class* `ProofGtNInsert'`. Demuestra que `Insert'` preserva el invariante `GtN`.

`ProofLtNInsert'` y `ProofGtNInsert'` son la implementación de lemas que demuestran que `Insert` mantiene los invariantes definidos por `LtN` y `GtN`.

Por ejemplo, `ProofLtNInsert'` es la implementación del Lema 3.3.1. Este predicado de prueba construye la evidencia de que `LtN (Insert' x a t o) n ~ 'True` dados `x`, `a`, `t`, `o` y asumiendo que se cumplen `CmpNat x n ~ 'LT` y `LtN t n ~ 'True` (restricciones requeridas en el contexto de la función). Es un valor que se construye recursivamente a partir de `Refl`, como se mostró en la Sección 2.9.

Este tipo de predicados de prueba, que construyen evidencia de igualdad de tipos, son utilizados aplicando la función `gcastWith`. Por ejemplo, en la siguiente instancia de `ProofIsBSTInsert'`

```
instance (l ~ 'ForkTree ll (Node ln lna) lr, o ~ CmpNat x ln,
  CmpNat x n ~ 'LT,
  ProofIsBSTInsert' x a l o, ProofLtNInsert' x a l n o) =>
  ProofIsBSTInsert' x a ('ForkTree ('ForkTree ll (Node ln lna) lr) (Node n a1) r) 'LT where
    proofIsBSTInsert' _ (ForkIsBSTT l pNode' r) _ =
      gcastWith (proofLtNInsert' pNode l (Proxy::Proxy n) po) $
      ForkIsBSTT (proofIsBSTInsert' pNode l po) pNode' r
        where
          po    = Proxy::Proxy o
          pNode = Proxy::Proxy (Node x a)
```

Código 4.37: Instancia de la *type class* `ProofIsBSTInsert'`. Aplicación de `proofLtNInsert'`.

La función `proofLtNInsert'` construye un valor cuyo tipo es `LtN (Insert' x a t o) n ~ 'True`. Luego, al aplicar `gcastWith` se incorpora al contexto la

igualdad de tipos `LtN (Insert' x a t o) n ∼ 'True`, que es la información que el compilador necesita para construir el nuevo término de prueba (dado que la inserción en este caso se realiza en la rama izquierda).

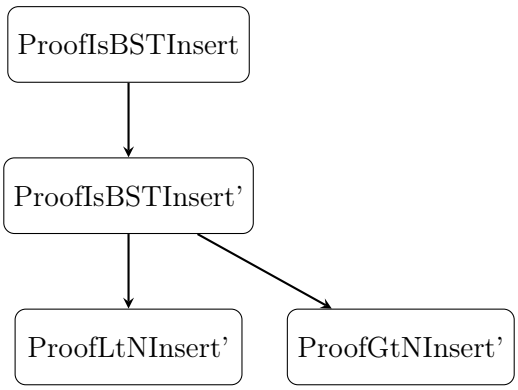


Figura 4.1: Relaciones entre las pruebas (todas ellas recursivas) para `ProofIsBSTInsert` del enfoque externalista (se omiten las dependencias recursivas).

En el caso de la operación de borrado, son necesarios otros lemas adicionales. Por ejemplo, cuando se elimina un nodo en el que sus subárboles son ambos no vacíos, se lo reemplaza por el nodo con mayor clave en el subárbol izquierdo (el elemento previo en el orden de las claves). Luego, es necesario probar que las claves del nuevo subárbol izquierdo son menores que el máximo.

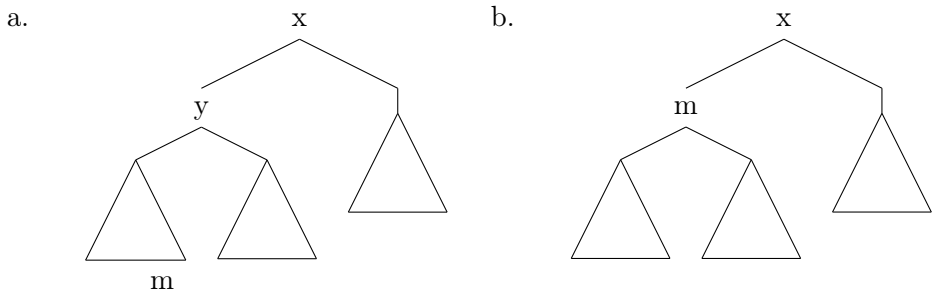


Figura 4.2: Para eliminar `y`, se lo reemplaza con el máximo `m` del subárbol izquierdo.

La clase `ProofIsBSTDelete` es la contraparte de `ProofIsBSTInsert` para el borrado y sigue un esquema similar, con el agregado de pruebas adicionales para

- probar que el subárbol izquierdo de `y` cumple el invariante `LtN` respecto de `x` luego de eliminar el máximo `m`

```
class ProofLtNMaxKeyDelete (t::Tree) (n::Nat) where
  proofLtNMaxKeyDelete :: (LtN t n ~ 'True) =>
    IsBSTT t → Proxy n → LtN (MaxKeyDelete t) n :~: 'True
```

Código 4.38: Borrar el nodo con clave máxima mantiene el invariante `LtN`.

- probar que la nueva raíz `m` (en lugar de `y`) es menor que `x`

```
class ProofLTMaxKey (t::Tree) (n::Nat) where
  proofLTMaxKey :: (LtN t n ~ 'True) =>
    IsBSTT t → Proxy n → CmpNat (MaxKey t) n :~: 'LT
```

Código 4.39: Si `LtN t n` $\sim$ `'True`, entonces la clave máxima de `t` es menor que `n`.

- en el caso en que el nodo `y` esté ubicado a la derecha de `x`, se necesitan pruebas `ProofGTMaxKey` y `ProofGtNMaxKeyDelete`, análogas a las anteriores (ahora respecto al invariante `GtN`).
- probar que la rama izquierda de `y` continúa siendo *BST* luego de que se elimina su máximo `m`

```
class ProofMaxKeyDeleteIsBST (t::Tree) where
  proofMaxKeyDeleteIsBST :: IsBSTT t → IsBSTT (MaxKeyDelete t)
```

Código 4.40: Borrar el nodo con clave máxima mantiene el invariante de *BST*.

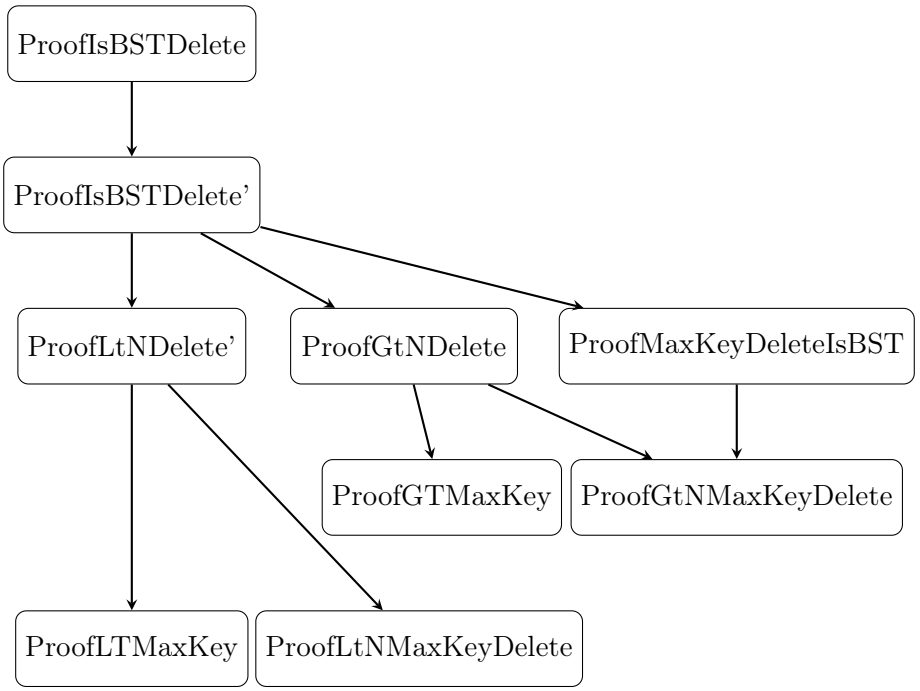


Figura 4.3: Relaciones entre las pruebas (todas ellas recursivas) del enfoque externalista (se omiten las dependencias recursivas).

La lógica de la inserción (así como la del borrado) es implementada al menos tres veces: una en la *type family* `Insert`, otra en la función `insert` y por último una vez más en cada función de prueba. Esto se observa al comparar por ejemplo instancias de `Insertable'` y `ProofIsBSTInsert'`:

```
instance Insertable' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT where
  type Insert' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT =
    'ForkTree ('ForkTree 'EmptyTree (Node x a) 'EmptyTree) (Node n a1) r
  insert' node (ForkITree _ node' r) _ =
    ForkITree (ForkITree EmptyITree node EmptyITree) node' r

instance ProofIsBSTInsert' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT where
  proofIsBSTInsert' _ (ForkIsBSTT _ pNode' r) _ =
    ForkIsBSTT (ForkIsBSTT EmptyIsBSTT pNode EmptyIsBSTT) pNode' r
      where
        pNode = Proxy::Proxy (Node x a)
```

Código 4.41: Duplicación de código entre las operaciones y las pruebas.

En cuanto al enfoque internalista, las pruebas se invocan directamente sobre los árboles `BST`, dado que no se mantiene ningún término de prueba.

Solamente se utilizan las restricciones a nivel de tipos con `LtN` y `GtN`, de modo que las únicas funciones de prueba que se necesitan son aquellas que verifican estas igualdades de tipos (e.g., `proofLtNInsert`).

Descartando las pruebas `ProofIsBSTInsert` y `ProofIsBSTDelete`, que no son necesarias con este enfoque, el resto de las pruebas y sus relaciones son muy similares al enfoque anterior, con diferencias sintácticas mínimas. Por ejemplo, comparando una instancia de `ProofGtNDelete'` para uno y otro enfoque

```
instance (l ~ 'ForkTree ll (Node ln la) lr, r ~ 'ForkTree rl (Node rn ra) rr,
  GtN l n ~ 'True, GtN r n ~ 'True,
  ProofGTMaxKey l n, ProofGtNMaxKeyDelete l n) =>
  ProofGtNDelete' x ('ForkTree ('ForkTree ll (Node ln la) lr) (Node n1 a1) r) n 'EQ where
  proofGtNDelete' _ (ForkIsBSTT l _ _) pn _ =
    gcastWith (proofGtNMaxKeyDelete l pn) $
    gcastWith (proofGTMaxKey l pn) Refl
```

Código 4.42: Instancia de `ProofGtNDelete'` para el enfoque externalista

```
instance (l ~ 'ForkTree ll (Node ln la) lr, r ~ 'ForkTree rl (Node rn ra) rr,
  GtN l n ~ 'True, GtN r n ~ 'True,
  ProofGTMaxKey l n, ProofGtNMaxKeyDelete l n) =>
  ProofGtNDelete' x ('ForkTree ('ForkTree ll (Node ln la) lr) (Node n1 a1) r) n 'EQ where
  proofGtNDelete' _ (ForkBST l _ _) pn _ =
    gcastWith (proofGtNMaxKeyDelete l pn) $
    gcastWith (proofGTMaxKey l pn) Refl
```

Código 4.43: Instancia de `ProofGtNDelete'` para el enfoque internalista.

la única diferencia apreciable es que en el primer caso la función de prueba es invocada sobre un término de prueba `IsBSTT`, mientras que en el segundo caso es directamente sobre un árbol `BST`.

En resumen, las principales características de cada enfoque a nivel de las pruebas se enuncian a continuación:

- El enfoque completamente externalista facilita la construcción automática del término de prueba.
- El enfoque externalista permite separar las pruebas de las operaciones en módulos diferentes, en virtud de que el término de prueba se encuentra desacoplado del término que representa al árbol.
- El enfoque internalista implementa la evidencia en los constructores de árboles `BST`, sin mantener un término de prueba por separado. Utiliza menos pruebas que en la versión externalista, pero la lógica de las operaciones se encuentra altamente acoplada a las funciones de prueba.

4.7 Desempeño

A diferencia del enfoque no seguro, en este caso la programación a nivel de tipos supone una carga adicional al proceso de compilación.

Por otro lado, el tiempo de compilación presenta una limitación al tamaño de los árboles que se pueden manejar en un tiempo razonable (e.g., la compilación de un árbol externalista o internalista con 50 nodos tarda cerca de un minuto y medio). Esta cantidad de nodos no es suficiente para apreciar diferencias significativas en los tiempos de ejecución de las operaciones de cada enfoque.

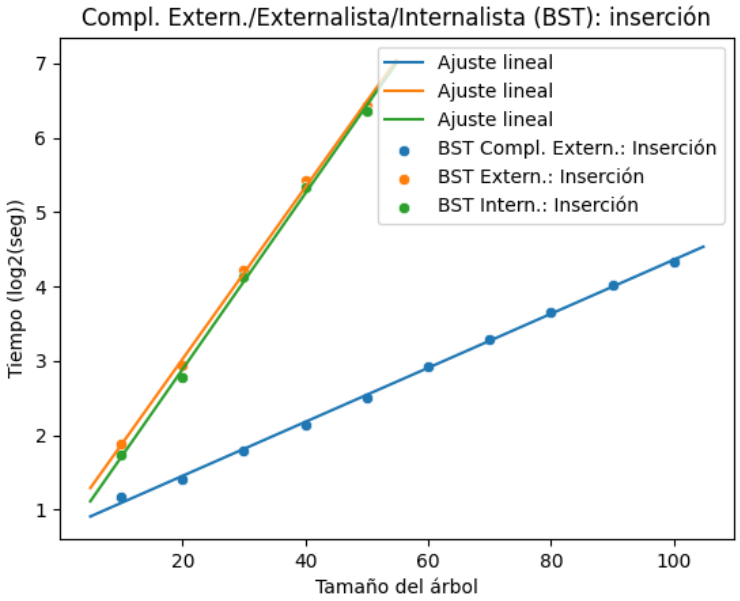


Figura 4.4: Tiempos de compilación en escala logarítmica de la inserción con los enfoques completamente externalista, externalista e internalista en árboles `BST` de distintos tamaños.

Por esto, la medida de desempeño elegida es el tiempo de compilación. Se midió el tiempo de compilación de programas que insertan/borran/buscan una clave en árboles con N cantidad de nodos para distintos valores de N, comparando las implementaciones de cada enfoque.

Utilizando una escala logarítmica (base 2) para los tiempos, en todos los casos (operaciones y enfoques) se observa una tendencia lineal. Esto indica que la tendencia original es exponencial (con base 2) en todos los casos.

Las operaciones del enfoque completamente externalista son las que compilan en menos tiempo, seguidas por las del enfoque internalista y por último las del enfoque externalista (en todas las operaciones).

El enfoque completamente externalista se desempeña significativamente mejor que los otros dos enfoques. Además, las tendencias muestran que la diferencia aumenta en función del tamaño del árbol.

Por otro lado, los enfoques externalista e internalista no muestran diferencias significativas, con una diferencia porcentual máxima (respecto al externalista) del 10.27% en los valores observados.

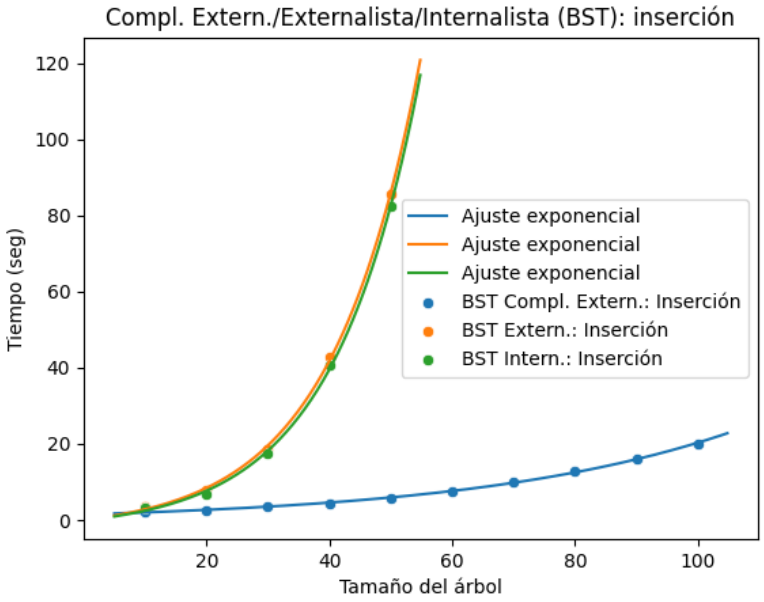


Figura 4.5: Tiempos de compilación de la inserción con los enfoques completamente externalista, externalista e internalista en árboles `BST` de distintos tamaños.

Es de esperar que el tiempo de compilación aumente con la cantidad de nodos, dado que el tamaño del tipo está directamente relacionado al tamaño del árbol. Sin embargo, la tendencia exponencial no condice con el crecimiento lineal de los tiempos de ejecución del enfoque no seguro. Tomando un ejemplo más simple, la búsqueda, que carece de funciones de prueba, también muestra este fenómeno: tiempos de ejecución lineales (logarítmicos para árboles `AVL`) en el enfoque no seguro y tiempos de compilación exponenciales en los enfoques seguros.

La comparación no es justa porque los procesos de compilación y ejecución son diferentes. Más aún, el ejemplo de la búsqueda muestra que las tendencias de los tiempos de ejecución no son extrapolables a los tiempos de compilación.

La explicación del carácter exponencial del tiempo de compilación queda fuera del alcance de este trabajo. Aunque es factible que esta explicación involucre al mecanismo de resolución de instancias, dado que todas las operaciones se implementaron a través de *type classes*.

En el Apéndice C se encuentran todos los valores de tiempos y en el Apéndice D las gráficas de estos. Por más detalles sobre el ajuste exponencial y la ejecución de los benchmarks, ver los Apéndices A y B.

4.8 Implementación guiada por tipos

Tomando como ejemplo la definición de la *type class* `Insertable'` de los enfoques externalistas

```
class Insertable' (x::Nat) (a::Type) (t::Tree) (o::Ordering) where
  type Insert' (x::Nat) (a::Type) (t::Tree) (o::Ordering) :: Tree
  insert' ::Node x a → ITree t → Proxy o → ITree (Insert' x a t o)
```

Código 4.44: Clase `Insertable'`. El tipo del resultado de `insert'` depende de la type family `Insert'`.

el tipo del resultado de la función `insert'` depende de la *type family* `Insert'`. En la instancia siguiente

```
instance Insertable' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT where
  type Insert' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT =
    'ForkTree ('ForkTree 'EmptyTree (Node x a) 'EmptyTree) (Node n a1) r
  insert' node' (ForkITree _ node r) _ =
    ForkITree (ForkITree EmptyITree node' EmptyITree) node r
```

Código 4.45: Ejemplo de instancia de `Insertable'` externalista. El tipo del árbol que se da como resultado en `insert'` debe coincidir con el tipo que retorna la *type family* `Insert'` para este caso.

el valor retornado por `insert'` debe tener el tipo `'ForkTree ('ForkTree 'EmptyTree (Node x a) 'EmptyTree) (Node n a1) r` (igual al resultado de la *type family* `Insert'` para esta instancia). De lo contrario ocurre un error en tiempo de compilación.

Más aún, la función de prueba también debe ser coherente con la *type family* `Insert'`, ya que su tipo también depende de ella:

```
class ProofIsBSTInsert' (x::Nat) (a::Type) (t::Tree) (o::Ordering) where
  proofIsBSTInsert' :: Proxy (Node x a) → IsBSTT t → Proxy o →
    IsBSTT (Insert' x a t o)
```

Código 4.46: Clase `ProofIsBSTInsert'`. El tipo del resultado de `proofIsBSTInsert'` depende de la type family `Insert'`.

Al mismo tiempo, la función de prueba debe retornar la evidencia de que la inserción respeta el invariante de *BST*. Si la función de prueba compila correctamente, se tiene la certeza de que los términos de prueba construidos se corresponden con árboles *BST*. Cualquier implementación de `Insertable'` que no cumpla el invariante de *BST* genera un error de compilación, pues en ese caso no es posible construir un término de prueba sin generar un error de compilación.

En base a estas observaciones es posible diseñar una metodología general de implementación guiada por tipos, que permite capturar errores de implementación en tiempo de compilación:

1. Implementar las estructuras de datos y sus invariantes.
2. Implementar las operaciones a nivel de tipos (*type families*).
3. Implementar las funciones de prueba que demuestran que las operaciones definidas anteriormente mantienen los invariantes.
4. Implementar las operaciones a nivel de valores correspondientes a las *type families* del punto 2.

En la transición del punto 2 al 3, las implementaciones de *type families* que no respeten los invariantes generan un error de compilación. Del mismo modo en la transición del punto 3 al 4, los errores en las implementaciones del punto 4 con respecto a los invariantes son capturados durante la compilación.

Sin embargo, es posible que las implementaciones de los puntos del 2 al 4 contengan un error que no sea capturado por el proceso de compilación. Por ejemplo, si en el caso de inserción a la izquierda se omite la inserción del nuevo nodo y se retorna el árbol original tanto en la *type family* `Insert'`, como en la función de prueba `proofIsBSTInsert'` y en la función `insert'`. En ese caso, no se detecta un incumplimiento en el invariante de *BST*, porque no se desordena ninguna clave. Además, las tres funciones son coherentes entre sí, a pesar de que la inserción no está implementada correctamente.

```
instance Insertable' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT where
  type Insert' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT =
    'ForkTree 'EmptyTree (Node n a1) r
  insert' node' (ForkITree EmptyITree node r) _ =
    ForkITree EmptyITree node r

instance (CmpNat x n ~ 'LT) =>
  ProofIsBSTInsert' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT where
  proofIsBSTInsert' _ (ForkIsBSTT EmptyIsBSTT pNode r) _ =
    ForkIsBSTT EmptyIsBSTT pNode r
```

Código 4.47: Implementación incorrecta de la inserción y la función de prueba. Sin embargo, no reporta errores de compilación.

En general, las funciones de prueba solamente verifican que una operación mantiene un invariante, que representa un aspecto específico de una estructura de datos. En este caso, solo se controla que la inserción preserve el orden de

los nodos en el árbol. Cualquier otra característica o propiedad de la estructura queda por fuera de este proceso de verificación. Por ejemplo, no se verifica si la inserción efectivamente inserta un nuevo nodo, o si se mantienen todos los nodos que tenía antes, o si la cantidad de nodos luego de la inserción es igual a la cantidad que tenía al principio más uno, etc.

A modo de conclusión, si bien existe la posibilidad de compilar correctamente código con errores, las restricciones a nivel de tipos permiten capturar errores de implementación relacionados a los invariantes en tiempo de compilación. De este modo, se tiene la certeza de que cualquier programa que compile correctamente no presentará estos errores durante su ejecución.

Capítulo 5

Implementación segura de árboles *AVL*

En este capítulo se extienden las implementaciones de árboles *BST* para incorporar invariantes de árboles *AVL*. Al igual que con los árboles `BST`, se implementa los `AVL` como un *GADT* con restricciones en los constructores.

Los árboles `AVL` implementados también almacenan elementos que contienen una clave a nivel de tipos y valores de distintos tipos. Se reutilizan el constructor de nodos y el de árboles `Tree` para la representación del árbol a nivel de tipos.

```
data Node :: Nat → Type → Type where
  Node :: a → Node k a
```

Código 5.1: Constructor de nodos.

```
data Tree :: Type where
  EmptyTree :: Tree
  ForkTree  :: Tree → n → Tree → Tree
```

Código 5.2: Constructor de árboles de tipo `Tree`.

A la verificación a nivel de tipos que antes se hacía en los `BST` acerca del orden de las claves, se agrega la del balance en las alturas. Es decir, al compilar el programa se verifica que las alturas de los árboles se encuentran balanceadas.

5.1 Condición de balance

Para verificar la condición de balance, es necesario calcular las alturas. Para esto se definen dos *type families*: una que calcula el máximo de dos números nautrales (las alturas de los subárboles izquierdo y derecho)

```
type family Max (n1::Nat) (n2::Nat) :: Nat where
  Max n1 n2 =
    (If (n1 <=? n2)
      n2  -- then
      n1  -- else
```

```
    )
```

Código 5.3: Máximo de dos números naturales a nivel de tipos.

y otra que calcula la altura de un árbol

```
type family Height (t::Tree) :: Nat where
  Height 'EmptyTree = 0
  Height ('ForkTree l (Node n a) r) =
    1 + Max (Height l) (Height r)
```

Código 5.4: Altura de un árbol a nivel de tipos.

Luego se define otra *type family* que decide si un par de números naturales, que representan las alturas de los subárboles izquierdo y derecho, difieren a lo sumo en una unidad:

```
type family BalancedHeights (h1::Nat) (h2::Nat) :: Bool where
  BalancedHeights 0   0  = 'True
  BalancedHeights 1   0  = 'True
  BalancedHeights h1  0  = 'False
  BalancedHeights 0   1  = 'True
  BalancedHeights 0   h2 = 'False
  BalancedHeights h1  h2 = BalancedHeights (h1-1) (h2-1)
```

Código 5.5: Determinar si las alturas de dos subárboles se encuentran balanceadas.

Para que un árbol sea *AVL* las alturas de cada pareja de subárboles en cada nodo debe evaluar `BalancedHeights` a `'True`.

5.1.1 Verificación a nivel de tipos

Como se mostró en la Sección 3.3.1, los algoritmos de inserción y borrado pueden desbalancear al árbol, dejándolo en una de seis posibles configuraciones. En cada caso, una aplicación adecuada de rotaciones en los nodos involucrados permite recuperar el balance perdido.

Para aplicar la rotación adecuada, es necesario identificar en cual configuración se encuentra el árbol. En el enfoque no seguro, se utilizaban las funciones `unbalancedState` (Código 3.3) y `balancedState` (Código 3.4), junto con dos tipos algebraicos de datos, para determinar si el árbol está desbalanceado y la rama (izquierda o derecha) con la mayor altura, respectivamente.

Para trasladar a tiempo de compilación la verificación del invariante de árboles *AVL*, se "elevan" las funciones a nivel de tipos en la forma de *type families*:

```
-- | - LeftUnbalanced: height(left subtree) = height(right subtree) + 2.
-- | - RightUnbalanced: height(right subtree) = height(leftt subtree) + 2.
-- | - NotUnbalanced: tree is not unbalanced.
data US = LeftUnbalanced | RightUnbalanced | NotUnbalanced

type family UnbalancedState (h1::Nat) (h2::Nat)::US where
  UnbalancedState 0 0   = 'NotUnbalanced
```

```
  UnbalancedState 1 0   = 'NotUnbalanced
  UnbalancedState 0 1   = 'NotUnbalanced
  UnbalancedState 2 0   = 'LeftUnbalanced
  UnbalancedState 0 2   = 'RightUnbalanced
  UnbalancedState h1 h2 = UnbalancedState (h1-1) (h2-1)
```

Código 5.6: `unbalancedState` a nivel de tipos.

```
-- | - LeftHeavy: height(left subtree) = height(right subtree) + 1.
-- | - RightHeavy: height(right subtree) = height(leftt subtree) + 1.
-- | - Balanced: height(left subtree) = height(right subtree).
data BS = LeftHeavy | RightHeavy | Balanced

type family BalancedState (h1::Nat) (h2::Nat) :: BS where
  BalancedState 0 0   = 'Balanced
  BalancedState 1 0   = 'LeftHeavy
  BalancedState 0 1   = 'RightHeavy
  BalancedState h1 h2 = BalancedState (h1-1) (h2-1)
```

Código 5.7: `balancedState` a nivel de tipos.

Los tipos algebraicos de datos `US` y `BS` mantienen su definición, pero también se elevan a nivel de tipos a través de la extensión del lenguaje *Data Kinds*: se crean los *kinds* `US` y `BS`, y los tipos `'NotUnbalanced`, `'LeftUnbalanced` y `'RightUnbalanced`, de *kind* `US`, y `'Balanced`, `'LeftHeavy` y `'RightHeavy`, de *kind* `BS`.

Por ejemplo, un árbol `t` en donde `UnbalancedState t = LeftUnbalanced`, significa que la altura de su rama izquierda es dos unidades mayor que la de la rama derecha. Es decir que se encuentra desbalanceado. Luego se inspecciona la diferencia en alturas de su rama izquierda. Si por ejemplo `t = ForkTree (ForkTree ll ln lr) node r` y `BalancedState (ForkTree ll ln lr) = LeftUnbalanced`, entonces la situación es como se muestra en la figura (en paréntesis las alturas de cada subárbol):

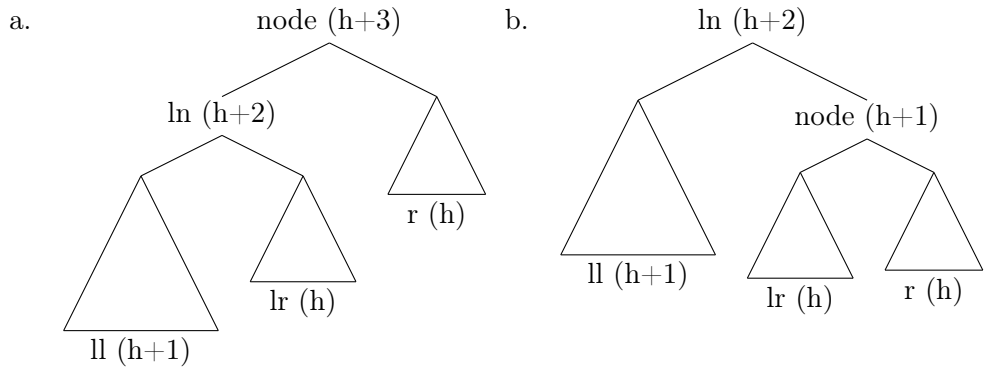


Figura 5.1: a. Árbol desbalanceado en su rama izquierda. b. Resultado de aplicar una rotación hacia la derecha.

5.2 Enfoque completamente externalista

El hecho de que los términos de prueba se encuentren desacoplados de los árboles permite reutilizar los términos y funciones de prueba de los árboles `BST` en la implementación de árboles `AVL`.

Para contemplar el invariante de balance en las alturas, se agrega un nuevo término de prueba junto con sus correspondientes funciones de prueba.

Por otro lado, aquellas situaciones en las que el árbol queda temporalmente desbalanceado luego de una inserción o un borrado se representan con un término de prueba particular.

5.2.1 Constructores

En el constructor de árboles `AVL` se reutilizan los *GADT* `Tree` e `IsBSTT` vistos en el Capítulo 4, a los que se agrega el término de prueba `IsBalancedT`, que verifica el balance en las alturas:

```
data IsBalancedT :: Tree → Type where
  EmptyIsBalancedT :: IsBalancedT 'EmptyTree
  ForkIsBalancedT  :: (BalancedHeights (Height l) (Height r) ~ 'True) =>
    IsBalancedT l → Proxy (Node n a) → IsBalancedT r → IsBalancedT ('ForkTree l
     (Node n a) r)
```

Código 5.8: Constructor de un término de prueba para árboles `AVL`. Contiene la evidencia a nivel de tipos de que las alturas están balanceadas.

Por otro lado, en los casos en que el árbol pierde el balance en las alturas debido a una inserción o un borrado, se observa el mismo patrón. Si se observan las seis configuraciones de desbalance presentadas en las Figuras 3.2 y 3.3, en todas se cumple que solo el nodo raíz tiene sus subárboles desbalanceados. Los subárboles izquierdo y derecho son `AVL`. Esto se implementa con el término de prueba `IsAlmostBalancedT`:

```
data IsAlmostBalancedT :: Tree → Type where
  EmptyIsAlmostBalancedT :: IsAlmostBalancedT 'EmptyTree
  ForkIsAlmostBalancedT  :: IsBalancedT l → Proxy (Node n a) → IsBalancedT r
    → IsAlmostBalancedT ('ForkTree l (Node n a) r)
```

Código 5.9: Constructor de un término de prueba para árboles `AlmostAVL`. Contiene la evidencia a nivel de tipos de que los subárboles izquierdo y derecho son `AVL`.

Para construir un `AVL` se necesita su representación a nivel de valores (de tipo `ITree t`), un término de prueba de tipo `IsBSTT t` (muestra que `t` es *BST*) y un término de prueba de tipo `IsBalancedT t` (muestra que `t` es *AVL*). El término de prueba `IsBalancedT` es la implementación de la definición de árbol *AVL* (ver 3.1.3).

```
data AVL :: Tree → Type where
  AVL :: ITree t → IsBSTT t → IsBalancedT t → AVL t
```

Código 5.10: Constructor externalista de árboles `AVL`. Reutiliza la implementación de `IsBST` e incorpora un nuevo término de prueba.

Como la estructura de los términos de prueba y del tipo `t` son isomorfas a la del árbol, el hecho de poder construir los términos de prueba asegura que las alturas del árbol se encuentran balanceadas (además de que las claves se encuentran ordenadas).

Se observa como ventaja de los enfoques externalistas la posibilidad de reutilizar el término de prueba `IsBSTT`. En otras palabras, para implementar árboles `AVL` tomando como base la implementación de árboles `BST`, basta extender el constructor de `BST` para incorporar el término de prueba `IsBalancedT`.

Nuevamente, para automatizar la construcción del término de prueba en el caso del enfoque completamente externalista, se delega esta tarea al sistema de inferencia de instancias de *type classes* definiendo una *type class* `IsBalancedC`:

```
class IsBalancedC (t :: Tree) where
  isBalancedT :: IsBalancedT t

instance IsBalancedC 'EmptyTree where
  isBalancedT = EmptyIsBalancedT
instance (IsBalancedC l, IsBalancedC r, BalancedHeights (Height l) (Height r) ~ 'True) =>
  IsBalancedC ('ForkTree l (Node n a) r) where
  isBalancedT = ForkIsBalancedT isBalancedT (Proxy::Proxy(Node n a)) isBalancedT

mkAVL :: (IsBSTC t, IsBalancedC t) => ITree t → AVL t
mkAVL t = AVL t isBSTT isBalancedT
```

Código 5.11: *Type class* para la construcción automática del término de prueba `IsBalancedT`. Constructor `mkAVL`.

5.2.2 Casos de error

Si se aplica `mkAVL` sobre un árbol que cumple con las restricciones de árbol *AVL*, entonces la compilación es exitosa. En cambio si se aplica a un árbol que no tiene todas sus claves ordenadas o que presenta algún desbalance en las alturas de sus árboles internos, entonces ocurre un error en tiempo de compilación. El error tiene su origen en no poder construir una instancia de `IsBSTC` o `IsBalancedC`. Por ejemplo:

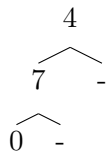


Figura 5.2: Ejemplo: no hay orden en las claves y hay subárboles con alturas desbalanceadas.

```
avlError = mkAVL $
  ForkITree (ForkITree
            (ForkITree
              EmptyITree (mkNode p0 'a') EmptyITree)
              (mkNode p7 4)
              EmptyITree)
            (mkNode p4 'f')
            EmptyITree
```

Código 5.12: Caso de errores de compilación: desorden en las claves y alturas desbalanceadas.

En este árbol

- la raíz tiene clave 4 y a su izquierda un nodo con clave 7 y
- las alturas de los subárboles izquierdo y derecho de la raíz difieren en 2.

Se obtienen los siguientes mensajes de error de compilación

```
• Key 7 is not smaller than key 4
• In the first argument of '($)', namely 'mkAVL'
In the expression:
mkAVL
$ ForkITree
(ForkITree
(ForkITree EmptyITree (mkNode p0 'a') EmptyITree) (mkNode p7 4)
EmptyITree)
(mkNode p4 'f') EmptyITree
...

• Couldn't match type ''False' with ''True'
arising from a use of 'mkAVL'
• In the expression:
mkAVL
$ ForkITree
(ForkITree
(ForkITree EmptyITree (mkNode p0 False) EmptyITree) (mkNode p7 4)
EmptyITree)
(mkNode p4 'f') EmptyITree
...
```

El primero fue generado por la *type family* `LtN` modificada para incorporar mensajes de error específicos.

En cuanto al segundo, al igual que con el ejemplo mostrado para `mkBST`, el mensaje de error indica que el origen del mismo se encuentra en la aplicación de `mkAVL`. La causa que reporta es, `Couldn't match type ''False' with ''True'`. Esto se debe a que la instancia de `IsBalancedT` requiere como parte de su contexto que `BalancedHeights (Height l) (Height r)` $\sim$ `'True`, pero en

este caso `BalancedHeights` evalúa a `'False`. Sin embargo, el mensaje no indica explícitamente que el error ocurre al intentar inferir las instancias de `IsBSTC` e `IsBalancedC`.

Así como se hizo con las *type families* `LtN`, `GtN` y `Member`, se incorporan mensajes de error específicos en la *type family* `BalancedHeights` y se agrega la clave del nodo como contexto:

```
type family BalancedHeights (h1 :: Nat) (h2 :: Nat) :: Bool where
  BalancedHeights 0   0   k = 'True
  BalancedHeights 1   0   k = 'True
  BalancedHeights h1  0   k = TypeError (Text "The left sub tree at node with key "
    :<>: ShowType k :<>: Text " has +2 greater height!")
  BalancedHeights 0   1   k = 'True
  BalancedHeights 0   h2  k = TypeError (Text "The right sub tree at node with key "
    :<>: ShowType k :<>: Text " has +2 greater height!")
  BalancedHeights h1  h2  k = BalancedHeights (h1-1) (h2-1) k
```

Código 5.13: *Type family* `BalancedHeights` con mensajes de error específicos.

Con esta modificación, el nuevo mensaje es

```
• The left sub tree at node with key 4 has +2 greater height!
• In the first argument of ‘($)’, namely ‘mkAVL’
In the expression:
mkAVL
$ ForkITree
(ForkITree
(ForkITree EmptyITree (mkNode p0 'a') EmptyITree) (mkNode p7 4)
EmptyITree)
(mkNode p4 'f') EmptyITree
...
```

De forma similar, es posible mostrar al usuario los subárboles que presentan el desbalance. La clave del nodo es suficiente para localizar el lugar en el árbol donde se encuentra el problema, dado que los árboles no admiten claves repetidas.

5.2.3 Rotaciones

La *type class* principal para rebalancear un árbol es `Balanceable`

```
class Balanceable (t::Tree) where
  type Balance (t::Tree) :: Tree
  balance :: ITree t → ITree (Balance t)
```

Código 5.14: *Type class* `Balanceable` en los enfoques externalistas.

que a su vez invoca las operaciones de una *type class* `Balanceable'` con el parámetro adicional `(us::US)`, que es el resultado de invocar `UnbalancedState` sobre el árbol.

```
class Balanceable' (t::Tree) (us::US) where
```

```
  type Balance' (t::Tree) (us::US) :: Tree
  balance' :: ITree t → Proxy us → ITree (Balance' t us)
```

Código 5.15: *Type class* `Balanceable'` en el enfoque externalista.

Por último, `Balanceable'` invoca a otra *type class* que implementa las rotaciones según la configuración en que se encuentre el árbol (que como ya se mencionó en la Sección 3.3.1, se determina con un valor de `US` y otro de `BS`):

```
class Rotateable (t::Tree) (us::US) (bs::BS) where
  type Rotate (t::Tree) (us::US) (bs::BS) :: Tree
  rotate :: ITree t → Proxy us → Proxy bs → ITree (Rotate t us bs)
```

Código 5.16: *Type class* `Rotateable` en los enfoques externalistas.

Cada instancia de `Rotateable` se corresponde con un caso distinto de desbalance del árbol.

Las implementaciones de las instancias de estas tres *type classes* coinciden con las del enfoque no seguro de la Sección 3.3.1. La diferencia es que los valores de `US` y `BS` ahora se manejan a nivel de tipos.

5.2.4 Inserción, Búsqueda y Borrado

En cuanto a la inserción, se modifican ligeramente las instancias de las clases `Insertable` (Código 4.13) e `Insertable'` (Código 4.14) de árboles `BST`. Además de insertar el nuevo nodo, se aplica la rotación necesaria para rebalancear el árbol:

```
instance (
  Balanceable ('ForkTree ('ForkTree 'EmptyTree (Node x a) 'EmptyTree) (Node n a1) r)) ⇒
  Insertable' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT where
  type Insert' x a ('ForkTree 'EmptyTree (Node n a1) r) 'LT =
    Balance ('ForkTree ('ForkTree 'EmptyTree (Node x a) 'EmptyTree) (Node n a1) r)
  insert' node (ForkITree _ node' r) _ =
    balance (ForkITree (ForkITree EmptyITree node EmptyITree) node' r)
```

Código 5.17: Ejemplo de instancia de la clase `Insertable'`. Las rotaciones se aplican a nivel de tipos y de valores. La instancia de `Balanceable` se define en el contexto.

Por otro lado, el borrado también es modificado para incorporar las rotaciones:

```
instance (o ~ CmpNat x rn, r ~ 'ForkTree rl (Node rn ra) rr,
  Deletable' x r o, Balanceable ('ForkTree l (Node n a1) (Delete' x r o))) ⇒
  Deletable' x ('ForkTree l (Node n a1) ('ForkTree rl (Node rn ra) rr)) 'GT where
  type Delete' x ('ForkTree l (Node n a1) ('ForkTree rl (Node rn ra) rr)) 'GT =
    Balance ('ForkTree l (Node n a1) (Delete' x ('ForkTree rl (Node rn ra) rr) (CmpNat x rn)))
  delete' px (ForkITree l node r) _ =
    balance $ ForkITree l node (delete' px r (Proxy::Proxy o))
```

Código 5.18: Ejemplo de instancia de la clase `Deletable'`. Las rotaciones se aplican a nivel de tipos y de valores. La instancia de `Balanceable` se define en el contexto.

Por último, la búsqueda no necesita ser modificada, ya que esta se define sobre el árbol de tipo `ITree t`, reutilizado por los árboles `AVL`.

5.3 Enfoque externalista

Se utilizan los mismos constructores `ITree` y `AVL` que en el enfoque presentado anteriormente, pero en este caso las operaciones inserción, búsqueda y borrado manejan árboles de tipo `AVL t` en lugar de `ITree t`.

Por otro lado, aquellas situaciones en las que el árbol queda temporalmente desbalanceado luego de una inserción o un borrado se representan con una estructura de datos particular.

5.3.1 Operaciones

Para cada operación se define una interfaz que reutiliza las operaciones del enfoque completamente externalista (del mismo modo que con los árboles `BST`):

```
insertAVL :: (Insertable x a t, ProofIsBSTInsert x a t, ProofIsBalancedInsert x a t)
    =>
  Proxy x -> a -> AVL t -> AVL (Insert x a t)
insertAVL (px::Proxy x) (a::a) (AVL t tIsBST tIsBalanced) =
  AVL (insert node t) (proofIsBSTInsert pNode tIsBST) (proofIsBalancedInsert pNode
    tIsBalanced)
    where node  = mkNode px a
          pNode = Proxy::Proxy (Node x a)

lookupAVL :: (t ~ 'ForkTree l (Node n a1) r, Member x t t ~ 'True, Lookupable x a t)
    => Proxy x -> AVL t -> a
lookupAVL p (AVL t _ _) = lookup p t

deleteAVL :: (Deletable x t, ProofIsBSTDelete x t, ProofIsBalancedDelete x t) =>
  Proxy x -> AVL t -> AVL (Delete x t)
deleteAVL px (AVL t tIsBST tIsBalanced) =
  AVL (delete px t) (proofIsBSTDelete px tIsBST) (proofIsBalancedDelete px
    tIsBalanced)
```

Código 5.19: Operaciones en el enfoque externalista

Las funciones de prueba relacionadas al invariante de *BST* (`proofIsBSTInsert` y `proofIsBSTDelete`) no son las mismas que fueron implementadas para los árboles `BST`. Dado que se modificaron las operaciones de inserción y borrado (para incorporar las rotaciones), es necesario reimplementar las funciones de prueba que demuestran que estas funciones también preservan el invariante de *BST*. Este punto se analiza con mayor profundidad en la Sección 5.5.

En la Sección 5.6 se evalúa el tiempo de compilación de cada operación sobre árboles de distintos tamaños.

5.4 Enfoque internalista

5.4.1 Constructores

A diferencia del enfoque externalista, el internalista no maneja un árbol `ITree`. Los invariantes son evaluados cada vez que un nuevo nodo es agregado al árbol.

```
data AVL :: Tree → Type where
  EmptyAVL :: AVL 'EmptyTree
  ForkAVL  :: (Show a, LtN l n ~ 'True, GtN r n ~ 'True,
    BalancedHeights (Height l) (Height r) ~ 'True) ⇒
    AVL l → Node n a → AVL r → AVL ('ForkTree l (Node n a) r)
```

Código 5.20: Constructor internalista de árboles AVL

Al igual que con el constructor internalista de árboles `BST`, se puede demostrar por inducción que los árboles construidos de esta forma son *AVL*.

Sin embargo, como se mencionó en la Sección 3.3.1, al insertar o eliminar nodos de un árbol `AVL`, este puede terminar en un estado de desbalance, que se definió como árbol *Almost AVL*. Se define una estructura de datos exclusiva para este tipo de árboles.

```
data AlmostAVL :: Tree → Type where
  AlmostAVL :: (Show a, LtN l n ~ 'True, GtN r n ~ 'True) ⇒
    AVL l → Node n a → AVL r → AlmostAVL ('ForkTree l (Node n a) r)
```

Código 5.21: Constructor internalista de árboles `AlmostAVL`

En los enfoques externalistas no fue necesario definir una nueva estructura de datos para este tipo de árboles; además se reutilizaron algunas de las estructuras de datos definidas para árboles `BST`. En cambio, en el enfoque internalista, no fue posible reutilizar ninguna de las estructura de datos de árboles `BST`.

5.4.2 Rotaciones

En el enfoque internalista, las operaciones para rebalancear no reciben como parámetro un `ITree`, sino un `AlmostAVL` y aplican las rotaciones necesarias para recuperar el balance en las alturas:

```
class Balanceable (t::Tree) where
  type Balance (t::Tree) :: Tree
  balance :: AlmostAVL t → AVL (Balance t)

class Balanceable' (t::Tree) (us::US) where
  type Balance' (t::Tree) (us::US) :: Tree
  balance' :: AlmostAVL t → Proxy us → AVL (Balance' t us)

class Rotateable (t::Tree) (us::US) (bs::BS) where
  type Rotate (t::Tree) (us::US) (bs::BS) :: Tree
  rotate :: AlmostAVL t → Proxy us → Proxy bs → AVL (Rotate t us bs)
```

Código 5.22: *Type classes* `Balanceable`, `Balanceable'` y `Rotateable` en el enfoque internalista.

El diseño es análogo al del enfoque externalista, con la diferencia que ahora los tipos reflejan la semántica de la operación. Por ejemplo, `balance` se aplica a un árbol `AlmostAVL` de tipo `AlmostAVL t` y retorna un árbol `AVL` de tipo `AVL (Balance t)`.

5.4.3 Inserción

En el enfoque internalista no se mantienen términos de prueba, de modo que las funciones `proofIsBSTInsert` y `proofIsBalancedInsert` ya no son necesarias. El método `insert` de la *type class* `Insertable` opera directamente sobre árboles `AVL`, no sobre árboles `ITree`.

```
class Insertable (x::Nat) (a::Type) (t::Tree) where
  type Insert (x::Nat) (a::Type) (t::Tree) :: Tree
  insert :: Node x a → AVL t → AVL (Insert x a t)
```

Código 5.23: `Insertable` internalista

Al igual que con los árboles `BST` internalistas, las demostraciones necesarias son implementadas en paralelo con las operaciones. En este caso, las demostraciones se realizan dentro de `insert`, ya que para construir un árbol `AVL`, es necesario demostrar que cumple los invariantes expresados en el constructor.

```
instance (r ~ 'ForkTree rl (Node rn rna) rr, o ~ CmpNat x rn,
  CmpNat x n ~ 'GT,
  Insertable' x a r o, Balanceable ('ForkTree l (Node n a1) (Insert' x a r o)),
  ProofGtNInsert' x a r n o) =>
  Insertable' x a ('ForkTree l (Node n a1) ('ForkTree rl (Node rn rna) rr)) 'GT where
    type Insert' x a ('ForkTree l (Node n a1) ('ForkTree rl (Node rn rna) rr)) 'GT =
      Balance ('ForkTree l (Node n a1) (Insert' x a ('ForkTree rl (Node rn rna) rr) (CmpNat x rn)))
  insert' node (ForkAVL l node' r) _ =
    gcastWith (proofGtNInsert' node r (Proxy::Proxy n) po) $
    balance $ AlmostAVL l node' (insert' node r po)
      where
        po = Proxy::Proxy o
```

Código 5.24: Ejemplo de instancia para `Insertable'` internalista. Uso de `AlmostAVL`.

El árbol que resulta de la inserción puede estar desbalanceado, por lo que el resultado de la llamada recursiva de `insert'` se retorna como el subárbol derecho de un árbol `AlmostAVL`. Luego, a este árbol se le aplica la rotación necesaria, mediante la función `balance`, para rebalancear las alturas si fuese necesario. Finalmente, se invoca a la función de pruba `proofGtNInsert'` para demostrar que se sigue cumpliendo el invariante de que las claves del subárbol derecho son mayores que la clave de la raíz.

Los árboles `AlmostAVL` se utilizan internamente en las funciones de inserción y borrado; el usuario de estas operaciones nunca manipula directamente este tipo de árboles.

5.4.4 Búsqueda y Borrado

El algoritmo de búsqueda es el mismo que para árboles `BST`, reimplementado para árboles `AVL`.

```
lookupAVL :: (t ~ 'ForkTree l (Node n a1) r, Member x t t ~ 'True,
    Lookupable x a t) => Proxy x → AVL t → a
lookupAVL = lookup
```

Código 5.25: Búsqueda internalista

Se mantiene a nivel de tipos el chequeo de la existencia de la clave buscada en el árbol, a través de la *type family* `Member`. Si la clave no se encuentra, se produce un error en tiempo de compilación, como se mostró en la Sección 4.3.3.

```
deleteAVL :: (Deletable x t) => Proxy x → AVL t → AVL (Delete x t)
deleteAVL = delete
```

Código 5.26: Borrado internalista

Al igual que con la inserción, se actualiza la lógica del borrado sobre árboles `BST`, de modo de incorporar las rotaciones.

```
instance (l ~ 'ForkTree ll (Node ln la) lr, o ~ CmpNat x ln,
  CmpNat x n ~ 'LT,
  Deletable' x l o, Balanceable ('ForkTree (Delete' x l o) (Node n a1) r),
  ProofLtNDelete' x l n o) =>
  Deletable' x ('ForkTree ('ForkTree ll (Node ln la) lr) (Node n a1) r) 'LT where
    type Delete' x ('ForkTree ('ForkTree ll (Node ln la) lr) (Node n a1) r) 'LT =
      Balance ('ForkTree (Delete' x ('ForkTree ll (Node ln la) lr) (CmpNat x ln))
               (Node n a1) r)
    delete' px (ForkAVL l node r) _ =
      gcastWith (proofLtNDelete' px l (Proxy::Proxy n) po) $
      balance $ AlmostAVL (delete' px l po) node r
        where
          po = Proxy::Proxy o
```

Código 5.27: Ejemplo de instancia de Deletable' para el enfoque internalista. Uso de `AlmostAVL`.

Como se muestra en el ejemplo, luego de cada llamada recursiva se aplica `balance` para recuperar el balance en las alturas.

En cuanto al desempeño de las operaciones, se evaluó el tiempo de compilación de cada operación sobre árboles de distintos tamaños, comparando árboles `BST` contra `AVL`, al igual que se hizo para el enfoque externalista. Se observan los mismos resultados que en aquel caso: tiempos de compilación exponenciales, con un mejor desempeño de los árboles `AVL`.

5.5 Pruebas

En la Sección 4.6 se comparó la implementación de las funciones de prueba sobre árboles `BST` en los distintos enfoques (externalistas e internalista). Las observa-

ciones realizadas entonces se mantienen para las pruebas sobre árboles `AVL`, por lo que en esta Sección se realiza la comparación de las funciones de prueba entre árboles `BST` y `AVL` para cada uno de los enfoques, comenzando con los enfoques externalistas.

En la Sección 5.2.1, los enfoques externalistas reutilizan la implementación del término de prueba `IsBSTT` de árboles `BST`. Sin embargo, al pasar a la implementación de árboles `AVL`, las operaciones de inserción y borrado fueron modificadas para aplicar las rotaciones. Por esta razón, es necesario reimplementar los predicados de prueba que demostraban que estas operaciones mantienen el invariante de *BST*. Por otro lado, también hay que demostrar que las rotaciones mantienen este invariante.

Por ejemplo, al comparar una instancia de `ProofIsBSTInsert'` para árboles `BST`

```
instance (l ~ 'ForkTree ll (Node ln lna) lr, o ~ CmpNat x ln,
  CmpNat x n ~ 'LT,
  ProofIsBSTInsert' x a l o, ProofLtNInsert' x a l n o) =>
  ProofIsBSTInsert' x a ('ForkTree ('ForkTree ll (Node ln lna) lr) (Node n a1) r) 'LT where
    proofIsBSTInsert' _ (ForkIsBSTT lIsBST pNode' rIsBST) _ =
      gcastWith (proofLtNInsert' pNode lIsBST (Proxy::Proxy n) po) $
      ForkIsBSTT (proofIsBSTInsert' pNode lIsBST po) pNode' rIsBST
        where
          po    = Proxy::Proxy o
          pNode = Proxy::Proxy (Node x a)
```

Código 5.28: Instancia de la clase `ProofIsBSTInsert'` para árboles `BST`.

con su contraparte para árboles `AVL`

```
instance (l ~ 'ForkTree ll (Node ln lna) lr, o ~ CmpNat x ln,
  CmpNat x n ~ 'LT,
  ProofIsBSTInsert' x a l o, ProofLtNInsert' x a l n o,
  ProofIsBSTBalance ('ForkTree (Insert' x a l o) (Node n a1) r)) =>
  ProofIsBSTInsert' x a ('ForkTree ('ForkTree ll (Node ln lna) lr) (Node n a1) r) 'LT where
    proofIsBSTInsert' _ (ForkIsBSTT lIsBST pNode' rIsBST) _ =
      gcastWith (proofLtNInsert' pNode lIsBST (Proxy::Proxy n) po) $
      proofIsBSTBalance $ ForkIsBSTT (proofIsBSTInsert' pNode lIsBST po) pNode' rIsBST
        where
          po    = Proxy::Proxy o
          pNode = Proxy::Proxy (Node x a)
```

Código 5.29: Instancia de la clase `ProofIsBSTInsert'` para árboles `AVL`.

se observa la incorporación de la función de prueba `proofIsBSTBalance`. Esto se debe a que la inserción sobre árboles `AVL` incorpora el uso de la función `balance` para rebalancear el árbol de ser necesario, como se mostró en el Código 5.27 sobre el borrado en árboles `AVL`.

En el caso de la operación de búsqueda, no fue necesaria una reimplementación de la misma ya que la lógica de la búsqueda en un árbol `BST` es la misma que en un `AVL`. Esta operación no tiene funciones de prueba asociadas, ya que no modifica el árbol. Sin embargo, es posible afirmar que en el caso de una operación sobre árboles `BST` que fuese reutilizada en árboles `AVL`, sus funciones de

prueba asociadas también podrían ser reutilizadas, debido al desacoplamiento de los términos de prueba respecto del constructor de los árboles.

Por otro lado, como al pasar de árboles `BST` a `AVL` se agregó un nuevo invariante (el balance en las alturas), hay que demostrar que las operaciones mantienen el nuevo invariante. Respecto al nuevo invariante, las pruebas que se definen son

- `ProofIsBalancedBalance` y `ProofIsBalancedRotate` para probar que `balance` y `rotate` recuperan el invariante del balance de las alturas.

- `ProofLtNBalance` y `ProofGtNBalance` para probar que `balance` preserva los invariantes definidos por `LtN` y `GtN` (y pruebas análogas para `balance'` y `rotate`).

En todos los casos, la relación entre las pruebas es similar. Por ejemplo, `ProofIsBSTBalance` invoca a `ProofIsBSTBalance'`, que a su vez invoca a `ProofIsBSTRotate`, replicando las dependencias entre `balance`, `balance'` y `rotate`.

A diferencia de las pruebas sobre la inserción o el borrado, ninguna de estas pruebas es recursiva, ya que las operaciones `balance`, `balance'` y `rotate` no lo son.

Por otro lado, al igual que ocurría con los árboles `BST`, hay una duplicación de código entre la lógica de cada operación y de sus funciones de prueba correspondientes.

En el enfoque internalista, al pasar de árboles `BST` a `AVL` hubo que implementar un nuevo constructor. Por esta razón fue necesario modificar tanto la implementación de las operaciones como de las funciones de prueba (independientemente de que en la inserción y el borrado hubo que incorporar a las rotaciones). Por esta razón, no fue posible reutilizar las funciones de prueba implementadas para árboles `BST`.

En cambio, no se utiliza el término de prueba `IsBalancedT`, de modo que no son necesarias las funciones de prueba como por ejemplo `ProofIsBalancedBalance`. Más allá de esto y del uso de `AlmostAVL`, el enfoque internalista tiene la misma organización de operaciones y pruebas que el enfoque externalista.

En resumen, las principales diferencias a nivel de las pruebas al pasar de árboles `BST` a `AVL` se enuncian a continuación:

- Los enfoques externalistas permiten la reutilización del término de prueba para árboles `BST` en árboles `AVL`.

- Además, permiten la reutilización de las funciones de prueba en aquellas operaciones sobre árboles `BST` que mantienen su implementación en árboles `AVL`.

- El enfoque internalista siempre necesita una reimplementación tanto de las operaciones como de las funciones de prueba, ya que al pasar de árboles `BST` a `AVL`, se utiliza un constructor de árboles diferente.

5.6 Desempeño

5.6.1 Comparación entre enfoques sobre árboles *AVL*

Como medida de desempeño se elige el tiempo de compilación, por las mismas razones que las expuestas en la discusión sobre el desempeño de los árboles `BST` de la Sección 4.7.

Al igual que con los árboles `BST`, las tendencias parecen ser exponenciales, dado que muestran una tendencia lineal en escala logarítmica. Nuevamente, el enfoque completamente externalista es el que mejor se desempeña, seguido por el internalista y el externalista en último lugar.

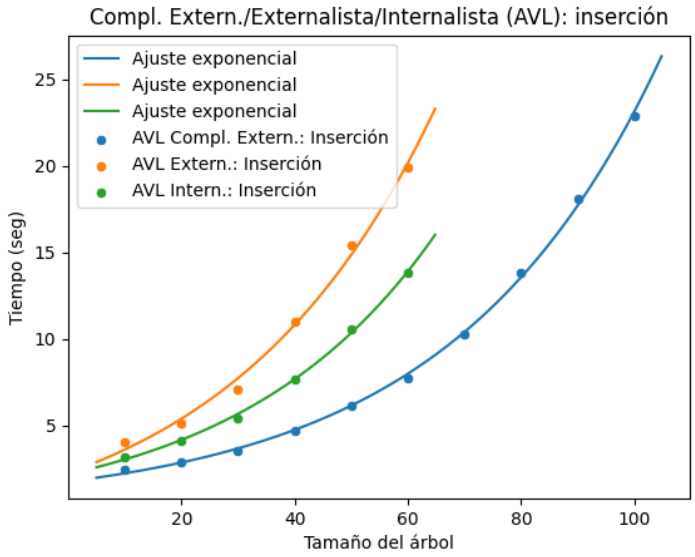


Figura 5.3: Tiempos de compilación de la inserción con los enfoques completamente externalista, externalista e internalista en árboles `AVL` de distintos tamaños.

En los árboles `AVL`, la diferencia del enfoque completamente externalista respecto a los otros dos enfoques es menor en términos relativos que la que se observó en árboles `BST`. Además, las diferencias relativas entre los enfoques externalista e internalista son mayores.

5.6.2 Comparación entre árboles *BST* y *AVL*

Se comparan los tiempos de compilación de cada operación sobre árboles `AVL` contra la misma operación en árboles `BST`.

En el enfoque completamente externalista, las operaciones sobre árboles `BST` son las que se desempeñan mejor. Las tendencias muestran que la diferencia aumenta conforme al tamaño del árbol. Por ejemplo, en la inserción las diferencias obtenidas no son significativas (una diferencia máxima de 2.80s para el árbol de 100 nodos). Sin embargo, en las tendencias exponenciales se observa que esta

diferencia puede aumentar en función del tamaño del árbol para árboles de más de 100 nodos:

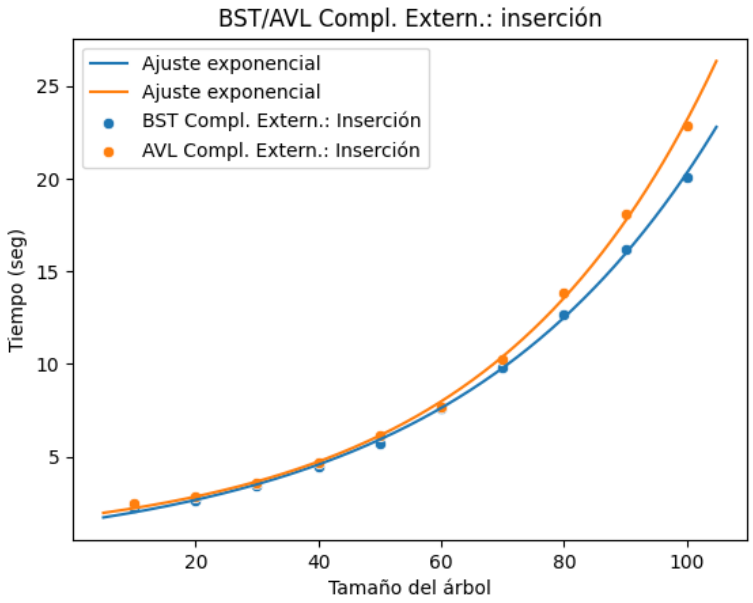


Figura 5.4: Tiempos de compilación de la inserción con el enfoque completamente externalista en árboles `BST` y `AVL` de distintos tamaños.

En este caso, la incorporación del balance en las alturas aumenta el tiempo de compilación. Esto es contrario a lo que ocurre en los tiempos de ejecución del enfoque no seguro: el balance en las alturas reduce los tiempos de acceso en el peor caso.

En este caso, la verificación de los invariantes se realiza automáticamente a través del mecanismo de inferencia de instancias de las *type classes*. Los árboles `BST` solamente verifican el orden en las claves, mientras que los `AVL` agregan la verificación del balance en las alturas. Era de esperar que el tiempo de compilación aumentase.

En cambio, los enfoques externalista e internalista muestran lo opuesto: un descenso significativo en los tiempos de compilación al pasar de árboles `BST` a `AVL`.

A diferencia del enfoque completamente externalista, estos enfoques requieren la implementación de funciones de prueba. La restricción del balance en las alturas significa más funciones de prueba y más operaciones (e.g., rotaciones), dado que hay un invariante más que considerar. Esto supone, en principio, un mayor costo de compilación, puesto que el compilador debe inferir las definiciones de las operaciones y funciones de prueba a través del mecanismo de inferencia de instancias de *type classes*.

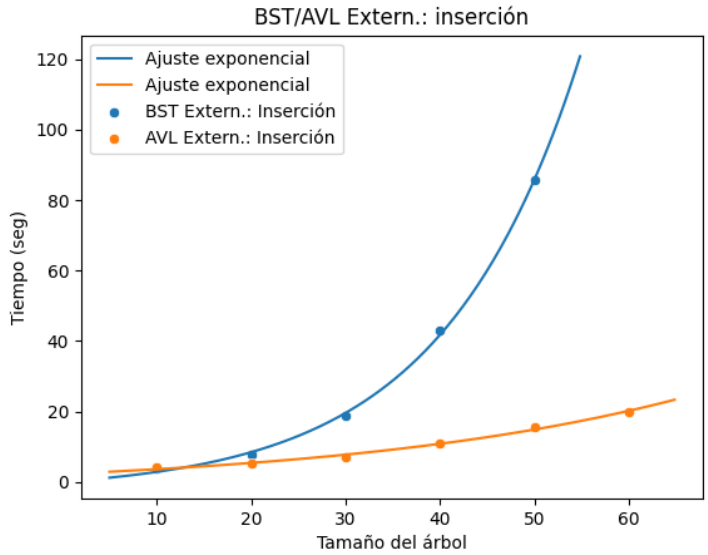


Figura 5.5: Tiempos de compilación de la inserción con el enfoque externalista en árboles `BST` y `AVL` de distintos tamaños.

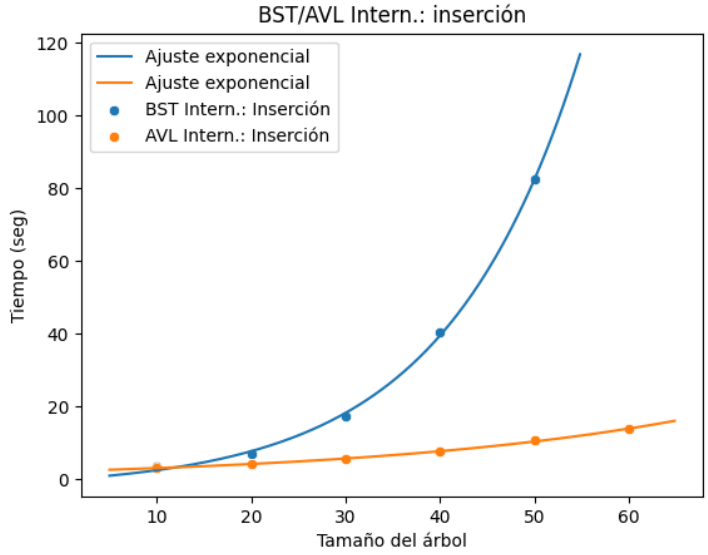


Figura 5.6: Tiempos de compilación de la inserción con el enfoque internalista en árboles `BST` y `AVL` de distintos tamaños.

A pesar de esto, los tiempos de compilación son menores en árboles `AVL` que en `BST`. Esto puede explicarse gracias a la restricción en el balance de las alturas. En efecto, la cantidad de instancias de *type classes* que deben ser inferidas para obtener las definiciones de las operaciones y funciones de prueba, depende directamente del recorrido en el árbol de la clave a insertar, buscar o borrar. Este

recorrido es menor en los árboles `AVL` que en los `BST` en el peor caso, con lo cual la cantidad de instancias por función que deben ser inferidas es menor en árboles `AVL`. Si bien la cantidad de funciones utilizadas en árboles `AVL` es mayor, los resultados muestran que aún así la carga de compilación es menor. Como se mencionó en la Sección 4.7, es factible que el tiempo de compilación pueda ser explicado a través del mecanismo de resolución de instancias. En ese caso, el mejor desempeño de los árboles `AVL`, en estos casos, se justifica porque requieren inferir un menor número de instancias.

En resumen, agregar el invariante de balance en las alturas incrementa el tiempo de compilación, pero simultáneamente reduce la carga que las operaciones y funciones de prueba suponen para el compilador.

Capítulo 6

Conclusión

Se implementaron las estructuras de árboles *BST* y *AVL* siguiendo distintos enfoques. En primer lugar con el enfoque no seguro, que no ofrece garantías de corrección del código. Luego con enfoques seguros (completamente externalista, externalista e internalista), que utilizan información a nivel de tipos, junto con el proceso de compilación para garantizar ciertas propiedades sobre los programas. A su vez, dentro de los enfoques seguros se distinguieron distintos enfoques según cómo se implementa esa información a nivel de tipos.

Sobre la verificación en tiempo de compilación de invariantes estructurales, los tres enfoques, el completamente externalista, el externalista y el internalista, permiten implementaciones de árboles *BST* y *AVL* que son seguros, en tanto que el proceso de compilación es utilizado como verificador de los invariantes estructurales. Esta verificación garantiza que se cumplirán las mismas propiedades en tiempo de ejecución.

Sin embargo, el uso de uno u otro enfoque tiene sus ventajas y desventajas según el aspecto que se analice (desempeño, reutilización de código, etc). Ningún enfoque es mejor que los otros en todos los aspectos.

En cuanto al desempeño, dado el costo computacional que supone la verificación de invariantes a nivel de tipos, se observan tiempos de compilación exponenciales en todos los casos. Esto limita las posibles aplicaciones prácticas de las implementaciones. Más allá de esto, en términos comparativos el enfoque completamente externalista muestra el mejor desempeño, lo cual es razonable dado que se reduce la cantidad de veces que los invariantes son verificados. Le sigue en desempeño el enfoque internalista y por último el externalista.

El mejor desempeño del enfoque completamente externalista se debe a que solamente el constructor de árboles realiza la verificación de los invariantes; ninguna de las operaciones, inserción o borrado, realiza verificaciones a nivel de tipos. En otras palabras, en el enfoque completamente externalista no se tiene la certeza de que las operaciones implementadas preserven los invariantes. Las implementaciones en los enfoques externalista e internalista tienen peor desempeño, pero el proceso de compilación garantiza que ninguna operación altera los invariantes.

La reducción en los tiempos de compilación en las operaciones de árboles *AVL*, comparadas con las operaciones sobre árboles *BST*, muestra la posibilidad de optimizaciones del tiempo de compilación. En particular, es posible que una

implementación de árboles *AVL* que incorpore a nivel de tipos el valor de las alturas en cada nodo, reduzca los tiempos de compilación en virtud de no requerir el cálculo de las alturas, como ocurre con la implementación actual.

Por otro lado, los enfoques externalistas permiten un diseño modular de la implementación, lo que facilita agregar restricciones (e.g.: implementar árboles *AVL* a partir de una implementación de árboles *BST*). También brinda la posibilidad de reutilizar código.

Un aspecto no cubierto fue la certificación de las operaciones en cuanto a su definición semántica. Las restricciones implementadas no certifican la corrección de las operaciones: no aseguran que la inserción inserte un nuevo nodo en el árbol manteniendo los nodos previos, que el borrado solamente quite un nodo, etc. Sin embargo, las mismas técnicas de programación a nivel de tipos podrían ser utilizadas para agregar invariantes que permitan definir especificaciones para cada operación.

Por último, queda como posible trabajo futuro la definición de una metodología en la cual se implementen las operaciones bajo un enfoque seguro y, luego de verificar que respetan los invariantes, se extraen de ellas operaciones análogas, borrando los tipos y las funciones de prueba utilizados para la verificación. Las operaciones para árboles *BST* y *AVL* implementadas con el enfoque no seguro fueron obtenidas de esta forma a partir de aquellas del enfoque completamente externalista. Este enfoque tiene la característica de que las funciones y términos de prueba se encuentran desacoplados de la lógica de las operaciones. Esto facilita la extracción de las operaciones. Un posible trabajo futuro es la definición de esta metodología.

Apéndice A

Especificaciones

Los benchmarks fueron ejecutados en una computadora con procesador Intel(R) Core(TM) i3-9100F CPU @ 3.60GHz de 4 núcleos y 16 GB de memoria RAM, sobre un sistema operativo Linux de 64 bits. Se utilizó la versión 3.7.6 de *python* y la 8.4.4 de *ghc*.

En el enfoque no seguro se midieron los tiempos de ejecución, mientras que en los enfoques seguros se midieron los tiempos de compilación.

En ambos casos se midió el tiempo de insertar/borrar/buscar una clave en árboles con N nodos, para distintos valores de N. Los árboles fueron construidos insertando nodos con claves consecutivas, 0, 1, 2, etc. De este modo los árboles `BST` tienen la estructura de una lista. Allí se insertó/borró/buscó el nodo con clave máxima, es decir el último nodo de la lista. Este es uno de los peores casos para las operaciones en árboles `BST`.

Solo se midió el tiempo asociado a la operación, sin tomar en cuenta el tiempo utilizado en construir el árbol de N nodos. Para los enfoques seguros, se separó en dos módulos la construcción del árbol y la aplicación de la operación sobre este. De este modo los módulos se compilan por separado y solo se midió el tiempo de compilación asociado a la operación.

Para cada valor de N y para cada operación, se ejecutaron/compilaron 16 instancias, de las cuales se quitan los valores atípicos (*outliers*). Finalmente se reporta el valor promedio.

Apéndice B

Aproximaciones

B.1 Enfoque no seguro

Los tiempos de ejecución de las operaciones en el enfoque no seguro se ajustan a funciones lineales de la forma $f(x) = a \cdot x + b$, mediante mínimos cuadrados.

Dados los valores medidos $y_1, ..., y_n$ y los valores dados por la función de aproximación $\hat{y_1}, ..., \hat{y_n}$, el método de mínimos cuadrados busca un mínimo (local) que minimice el error medido como $Error = \sum_{i=1}^{n} (y_i - \hat{y_i})^2$. También se toma como medida de error el *Root Mean Squared Error (RMSE)*: $\sqrt{\frac{\sum_{i=1}^{n} (y_i - \hat{y_i})^2}{n}}$.

Enfoque no seguro						
	Inserción		*Borrado*		*Búsqueda*	
-	`BST`	`AVL`	`BST`	`AVL`	`BST`	`AVL`
a	6.83×10^{-7}	1.95×10^{-6}	6.99×10^{-7}	2.00×10^{-6}	1.74×10^{-8}	-
b	3.82×10^{-5}	-4.82×10^{-4}	-9.40×10^{-5}	-5.29×10^{-4}	1.58×10^{-5}	-
Error	1.47×10^{-8}	5.51×10^{-6}	6.79×10^{-7}	9.11×10^{-6}	6.72×10^{-10}	-
RMSE	3.83×10^{-5}	7.42×10^{-4}	2.61×10^{-4}	9.54×10^{-4}	8.19×10^{-6}	-

Tabla B.1: Valores de a y b en el ajuste a la función lineal $f(x) = a \cdot x + b$. Valores de error. Valores de *Root Mean Squared Error (RMSE)*.

B.2 Enfoques seguros

Los tiempos de compilación de las operaciones en los enfoques seguros se ajustan a funciones exponenciales de la forma $f(x) = a \cdot 2^{b \cdot x} + c$, mediante el algoritmo de Levenberg-Marquardt para el ajuste con mínimos cuadrados de funciones no lineales[Lev44], implementado en el método `curve_fit` de la biblioteca `scipy` de Python[1].

[1] The SciPy community. *SciPy: Optimization.* curve_fit. Sphinx. Ver. 1.7.1. Mar. de 2021. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html (visitado 19-06-2021). En línea.

Enfoque completamente externalista						
	Inserción		*Borrado*		*Búsqueda*	
-	`BST`	`AVL`	`BST`	`AVL`	`BST`	`AVL`
a	1.997	1.530	2.089	1.752	2.088	1.528
b	0.0338	0.0390	0.0337	0.0377	0.0335	0.0390
c	-0.5257	0.214	-0.748	-0.240	-0.739	0.082
Error	2.91×10^{-1}	4.64×10^{-1}	3.60×10^{-1}	7.85×10^{-1}	2.97×10^{-1}	6.45×10^{-1}
RMSE	1.70×10^{-1}	2.15×10^{-1}	1.90×10^{-1}	2.80×10^{-1}	1.72×10^{-1}	2.54×10^{-1}

Tabla B.2: Valores de a, b y c en el ajuste a la función exponencial $f(x) = a \cdot 2^{b \cdot x} + c$. Valores de error. Valores de *Root Mean Squared Error (RMSE)*.

Enfoque externalista						
	Inserción		*Borrado*		*Búsqueda*	
-	`BST`	`AVL`	`BST`	`AVL`	`BST`	`AVL`
a	2.842	4.333	2.905	3.577	2.390	3.517
b	0.099	0.039	0.099	0.043	0.102	0.043
c	-2.831	-2.100	-4.014	-1.833	-2.797	-1.829
Error	3.34	1.13	2.00	4.53×10^{-1}	2.88	5.00×10^{-1}
RMSE	8.18×10^{-1}	4.34×10^{-1}	6.32×10^{-1}	2.75×10^{-1}	7.58×10^{-1}	2.89×10^{-1}

Tabla B.3: Valores de a, b y c en el ajuste a la función exponencial $f(x) = a \cdot 2^{b \cdot x} + c$. Valores de error. Valores de *Root Mean Squared Error (RMSE)*.

Enfoque internalista						
	Inserción		*Borrado*		*Búsqueda*	
-	`BST`	`AVL`	`BST`	`AVL`	`BST`	`AVL`
a	2.535	2.559	2.516	2.597	2.293	2.492
b	0.101	0.041	0.102	0.041	0.103	0.042
c	-2.672	-0.385	-3.187	-1.092	-2.892	-0.962
Error	3.04	1.25×10^{-1}	2.67	1.70×10^{-1}	3.40	9.81×10^{-2}
RMSE	7.80×10^{-1}	1.44×10^{-1}	7.30×10^{-1}	1.68×10^{-1}	8.24×10^{-1}	1.28×10^{-1}

Tabla B.4: Valores de a, b y c en el ajuste a la función exponencial $f(x) = a \cdot 2^{b \cdot x} + c$. Valores de error. Valores de *Root Mean Squared Error (RMSE)*.

Apéndice C

Resultados de benchmarks

Implementación no segura						
	Inserción		*Borrado*		*Búsqueda*	
N	`BST`	`AVL`	`BST`	`AVL`	`BST`	`AVL`
2^6	5.41×10^{-5}	$\mathbf{5.33 \times 10^{-5}}$	$\mathbf{4.38 \times 10^{-5}}$	3.23×10^{-4}	9.41×10^{-6}	$\mathbf{7.16 \times 10^{-6}}$
2^7	1.34×10^{-4}	$\mathbf{8.71 \times 10^{-5}}$	$\mathbf{7.90 \times 10^{-5}}$	8.36×10^{-5}	7.83×10^{-6}	$\mathbf{3.06 \times 10^{-6}}$
2^8	$\mathbf{1.71 \times 10^{-4}}$	1.95×10^{-4}	$\mathbf{1.59 \times 10^{-4}}$	1.69×10^{-4}	1.31×10^{-5}	$\mathbf{2.54 \times 10^{-6}}$
2^9	3.67×10^{-4}	$\mathbf{3.44 \times 10^{-4}}$	3.49×10^{-4}	$\mathbf{3.24 \times 10^{-4}}$	2.21×10^{-5}	$\mathbf{2.68 \times 10^{-6}}$
2^{10}	7.28×10^{-4}	$\mathbf{7.07 \times 10^{-4}}$	$\mathbf{6.79 \times 10^{-4}}$	8.45×10^{-4}	3.87×10^{-5}	$\mathbf{2.81 \times 10^{-6}}$
2^{11}	$\mathbf{1.43 \times 10^{-3}}$	1.95×10^{-3}	$\mathbf{1.39 \times 10^{-3}}$	4.87×10^{-3}	6.34×10^{-5}	$\mathbf{2.74 \times 10^{-6}}$
2^{12}	$\mathbf{2.89 \times 10^{-3}}$	8.06×10^{-3}	$\mathbf{2.80 \times 10^{-3}}$	5.60×10^{-3}	1.00×10^{-4}	$\mathbf{2.85 \times 10^{-6}}$
2^{13}	$\mathbf{5.62 \times 10^{-3}}$	1.68×10^{-2}	$\mathbf{5.47 \times 10^{-3}}$	1.67×10^{-2}	1.62×10^{-4}	$\mathbf{2.92 \times 10^{-6}}$
2^{14}	$\mathbf{1.13 \times 10^{-2}}$	3.16×10^{-2}	$\mathbf{1.06 \times 10^{-2}}$	3.13×10^{-2}	2.91×10^{-4}	$\mathbf{2.88 \times 10^{-6}}$
2^{15}	$\mathbf{2.23 \times 10^{-2}}$	6.31×10^{-2}	$\mathbf{2.31 \times 10^{-2}}$	6.55×10^{-2}	5.86×10^{-4}	$\mathbf{2.74 \times 10^{-6}}$

Tabla C.1: Tiempos de ejecución de cada operación en el enfoque no seguro. Comparación entre árboles `BST` y `AVL` del tiempo de ejecución de cada operación sobre árboles de tamaño N.

Implementación completamente externalista						
	Inserción		*Borrado*		*Búsqueda*	
N	`BST`	`AVL`	`BST`	`AVL`	`BST`	`AVL`
10	**2.25**	2.46	**2.16**	2.3	**2.15**	2.35
20	**2.66**	2.86	**2.59**	2.78	**2.58**	2.73
30	**3.45**	3.55	**3.37**	3.46	**3.36**	3.39
40	**4.43**	4.66	**4.43**	4.67	**4.4**	4.52
50	**5.69**	6.13	**5.76**	6.19	**5.77**	6.01
60	**7.59**	7.71	**7.66**	7.81	7.56	**7.47**
70	**9.82**	10.25	**9.99**	10.51	**9.91**	10.06
80	**12.67**	13.82	**12.98**	14.26	**12.8**	13.74
90	**16.22**	18.11	**16.7**	18.73	**16.43**	17.96
100	**20.09**	22.89	**20.62**	23.33	**20.32**	22.66

Tabla C.2: Tiempos de compilación de cada operación en el enfoque completamente externalista. Comparación entre árboles `BST` y `AVL` del tiempo de compilación de cada operación sobre árboles de tamaño N.

Implementación externalista						
	Inserción		*Borrado*		*Búsqueda*	
N	`BST`	`AVL`	`BST`	`AVL`	`BST`	`AVL`
10	**3.7**	4.03	**2.44**	3.26	**2.92**	3.17
20	7.69	**5.09**	6.8	**4.48**	6.3	**4.4**
30	18.66	**7.1**	17.98	**6.51**	16.48	**6.28**
40	42.85	**11.0**	41.76	**9.99**	38.91	**9.76**
50	85.76	**15.45**	85.0	**14.39**	79.85	**14.05**
60	-	**19.9**	-	**19.29**	-	**18.78**

Tabla C.3: Tiempos de compilación de cada operación en el enfoque externalista. Comparación entre árboles `BST` y `AVL` del tiempo de compilación de cada operación sobre árboles de tamaño N.

Implementación internalista						
	Inserción		*Borrado*		*Búsqueda*	
N	`BST`	`AVL`	`BST`	`AVL`	`BST`	`AVL`
10	3.32	**3.14**	2.7	**2.49**	2.68	**2.48**
20	6.9	**4.12**	6.43	**3.48**	5.85	**3.42**
30	17.4	**5.44**	16.82	**4.73**	15.53	**4.8**
40	40.52	**7.69**	39.86	**7.09**	37.54	**6.95**
50	82.39	**10.57**	81.8	**9.95**	77.06	**9.82**
60	-	**13.81**	-	**13.17**	-	**13.06**

Tabla C.4: Tiempos de compilación de cada operación en el enfoque internalista. Comparación entre árboles `BST` y `AVL` del tiempo de compilación de cada operación sobre árboles de tamaño N.

Apéndice D

Gráficas

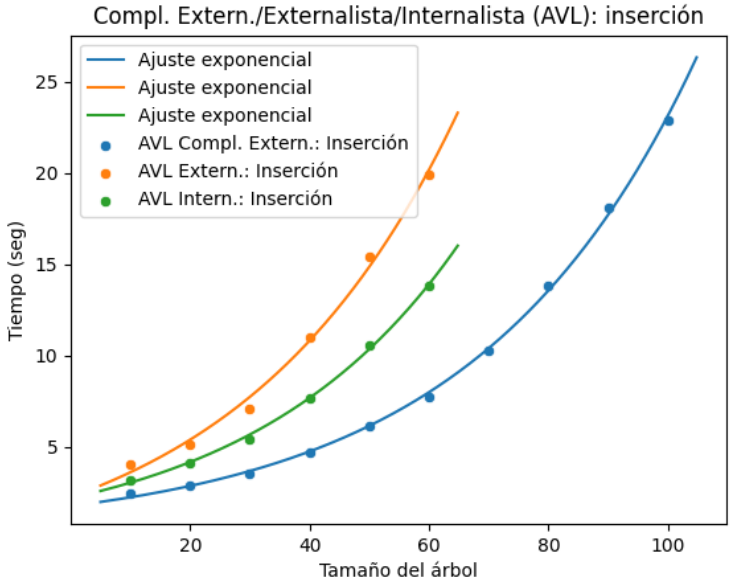


Figura D.1: Tiempos de compilación de la inserción con los enfoques completamente externalista, externalista e internalista en árboles `AVL` de distintos tamaños.

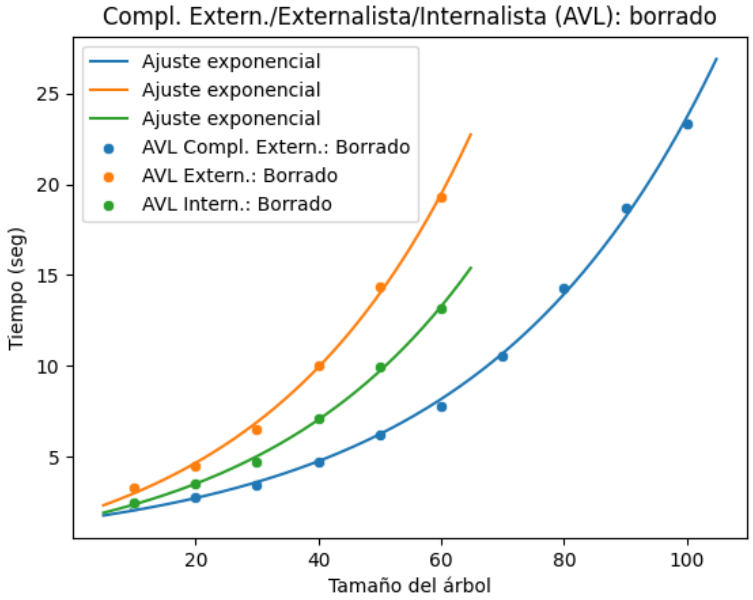


Figura D.2: Tiempos de compilación del borrado con los enfoques completamente externalista, externalista e internalista en árboles `AVL` de distintos tamaños.

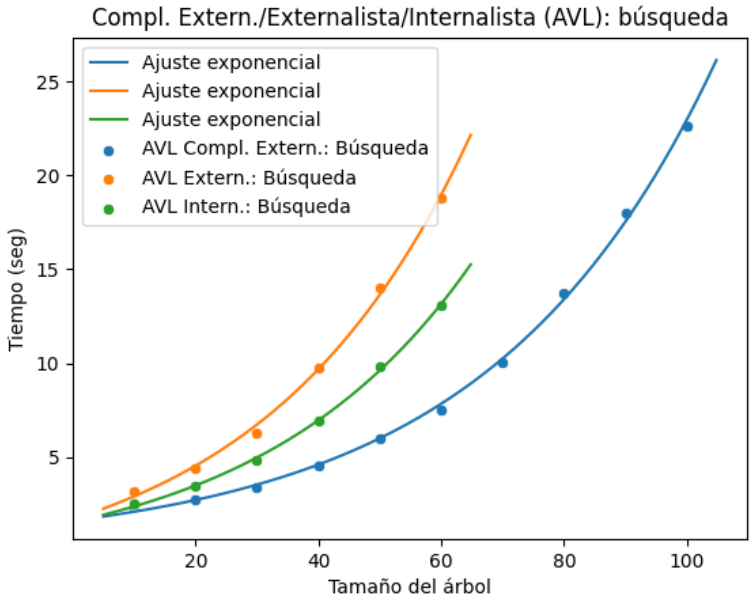


Figura D.3: Tiempos de compilación de la búsqueda con los enfoques completamente externalista, externalista e internalista en árboles `AVL` de distintos tamaños.

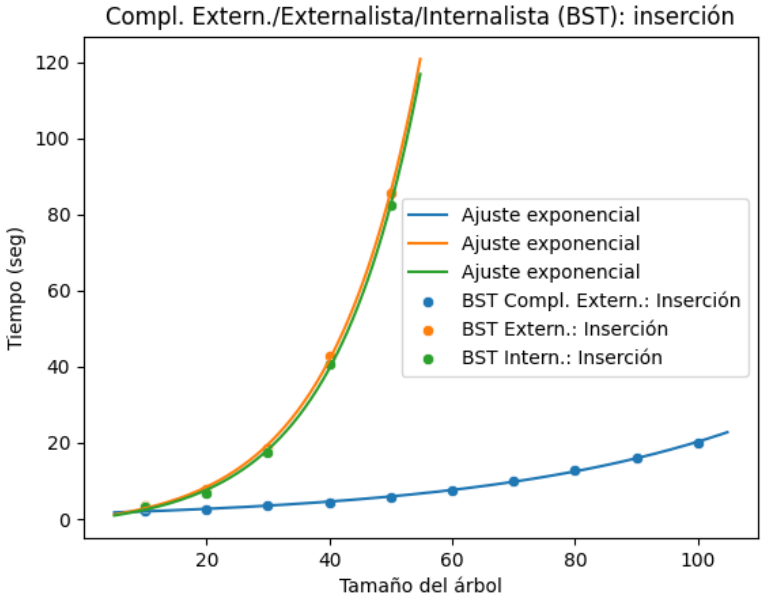


Figura D.4: Tiempos de compilación de la inserción con los enfoques completamente externalista, externalista e internalista en árboles `BST` de distintos tamaños.

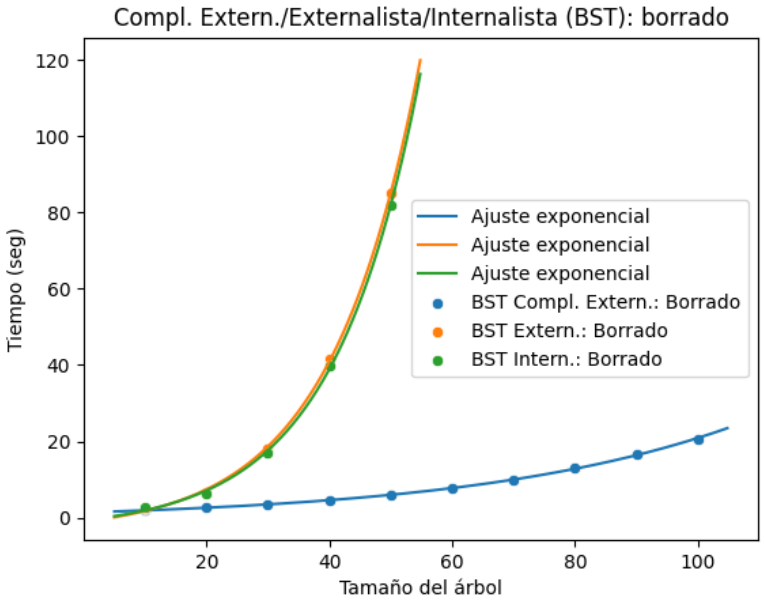


Figura D.5: Tiempos de compilación del borrado con los enfoques completamente externalista, externalista e internalista en árboles `BST` de distintos tamaños.

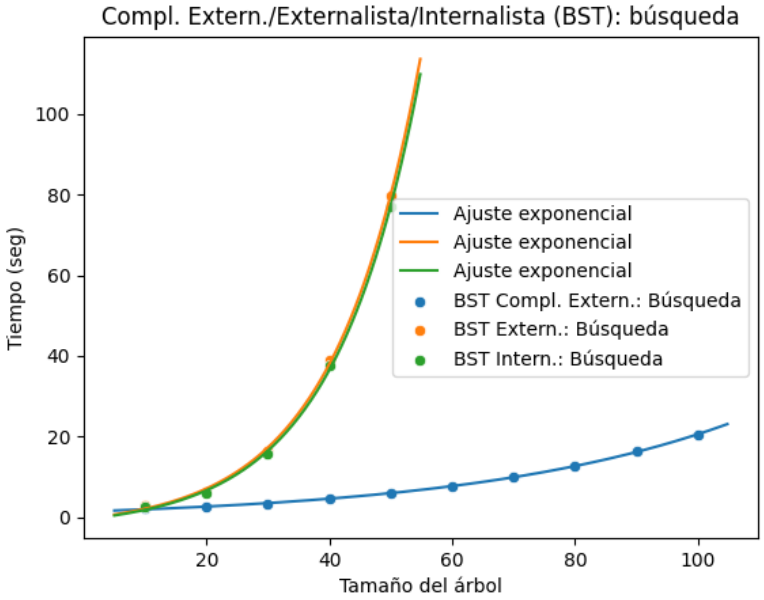


Figura D.6: Tiempos de compilación de la búsqueda con los enfoques completamente externalista, externalista e internalista en árboles `BST` de distintos tamaños.

D.1 Enfoque no seguro

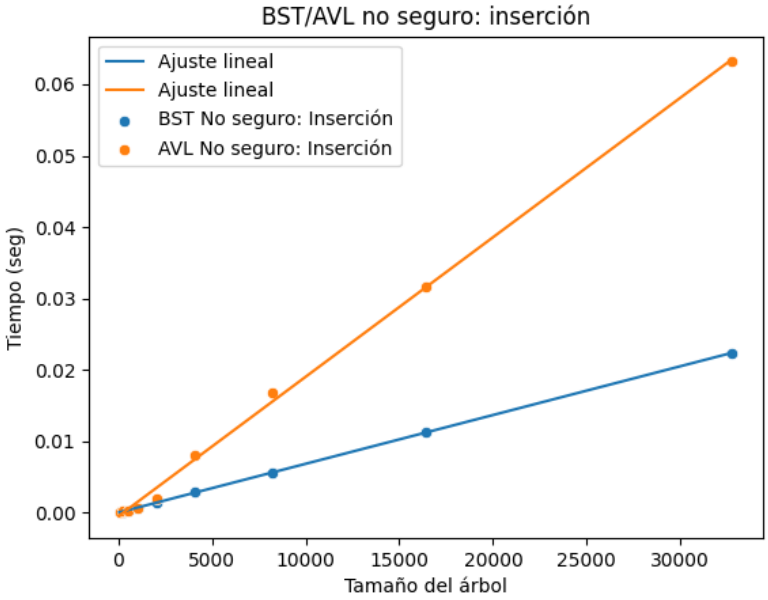


Figura D.7: Tiempos de ejecución de la inserción con el enfoque no seguro en árboles `BST` y `AVL` de distintos tamaños.

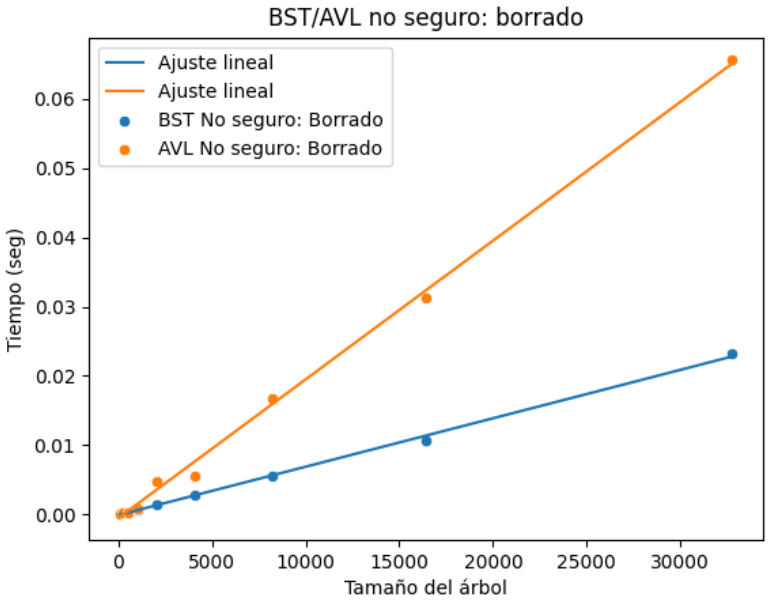


Figura D.8: Tiempos de ejecución del borrado con el enfoque no seguro en árboles `BST` y `AVL` de distintos tamaños.

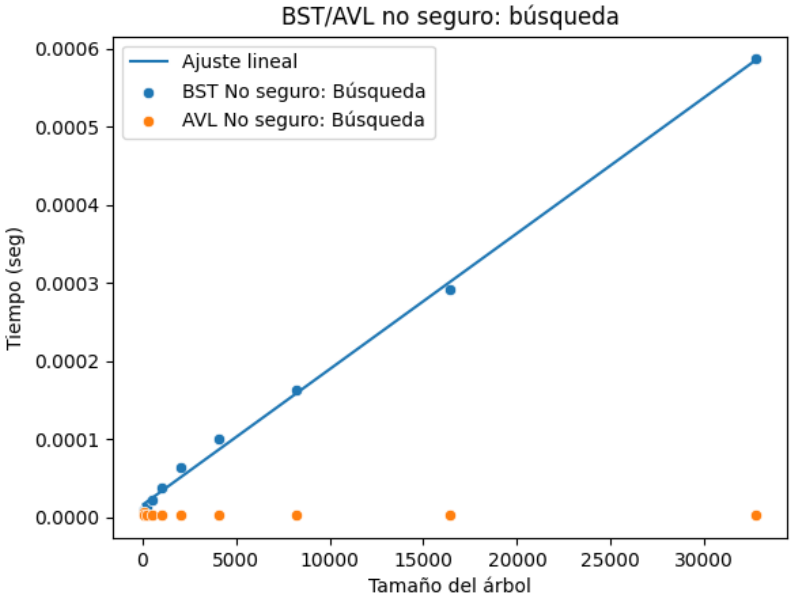


Figura D.9: Tiempos de ejecución de la búsqueda con el enfoque no seguro en árboles `BST` y `AVL` de distintos tamaños.

D.2 Enfoque completamente externalista

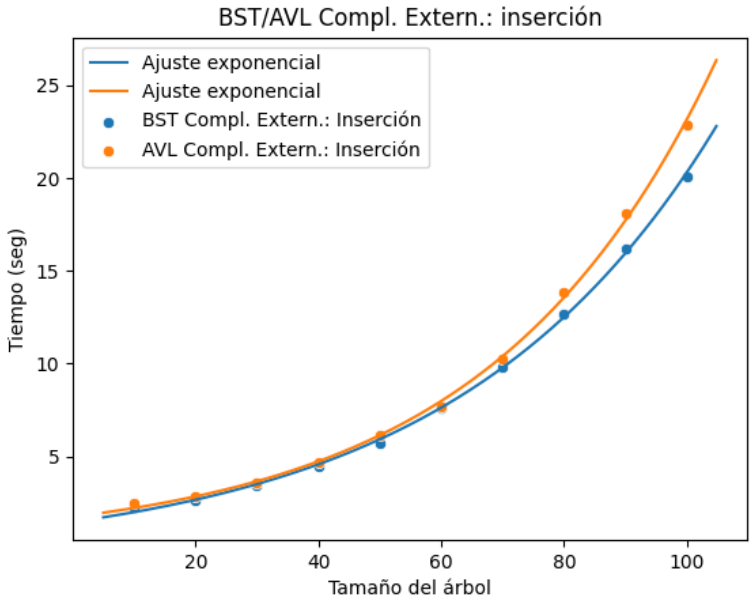


Figura D.10: Tiempos de compilación de la inserción con el enfoque completamente externalista en árboles `BST` y `AVL` de distintos tamaños.

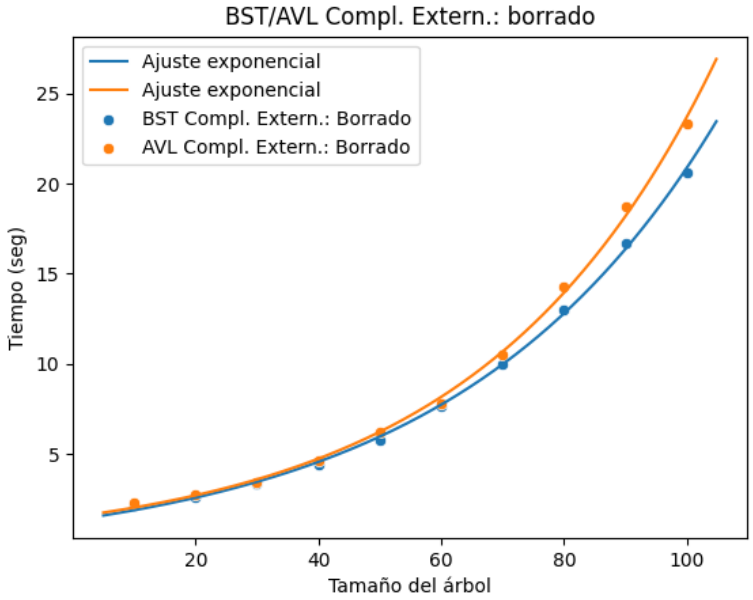


Figura D.11: Tiempos de compilación del borrado con el enfoque completamente externalista en árboles `BST` y `AVL` de distintos tamaños.

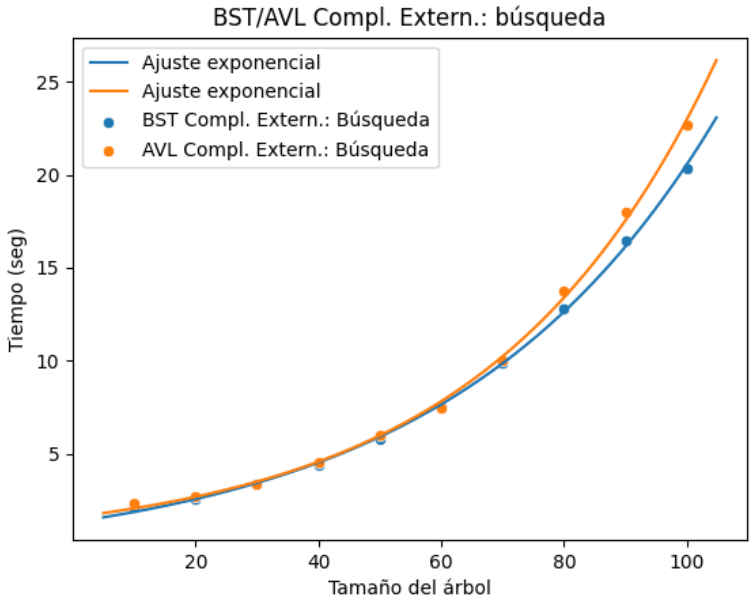


Figura D.12: Tiempos de compilación de la búsqueda con el enfoque completamente externalista en árboles `BST` y `AVL` de distintos tamaños.

D.3 Enfoque externalista

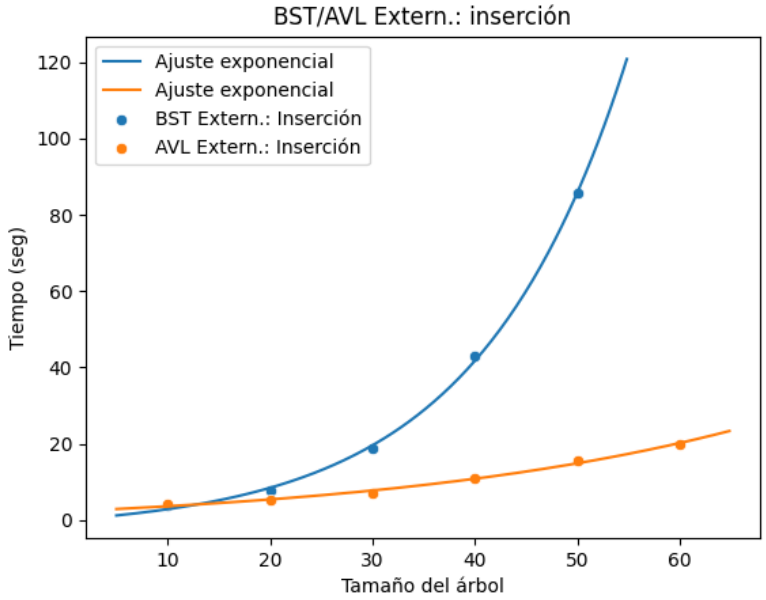


Figura D.13: Tiempos de compilación de la inserción con el enfoque externalista en árboles `BST` y `AVL` de distintos tamaños.

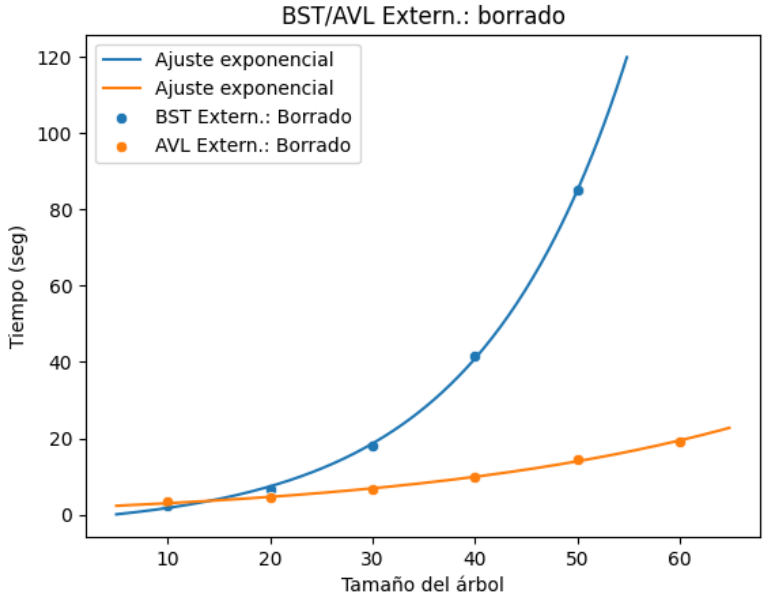


Figura D.14: Tiempos de compilación del borrado con el enfoque externalista en árboles `BST` y `AVL` de distintos tamaños.

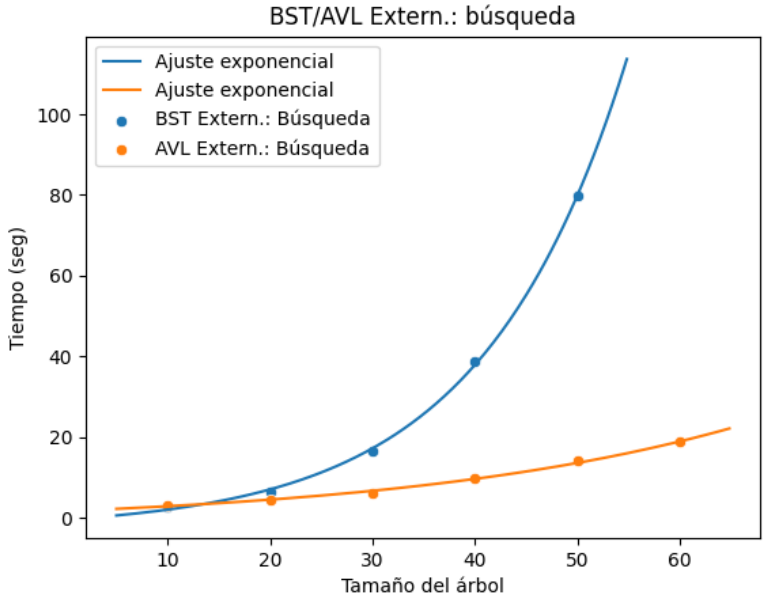


Figura D.15: Tiempos de compilación de la búsqueda con el enfoque externalista en árboles `BST` y `AVL` de distintos tamaños.

D.4 Enfoque internalista

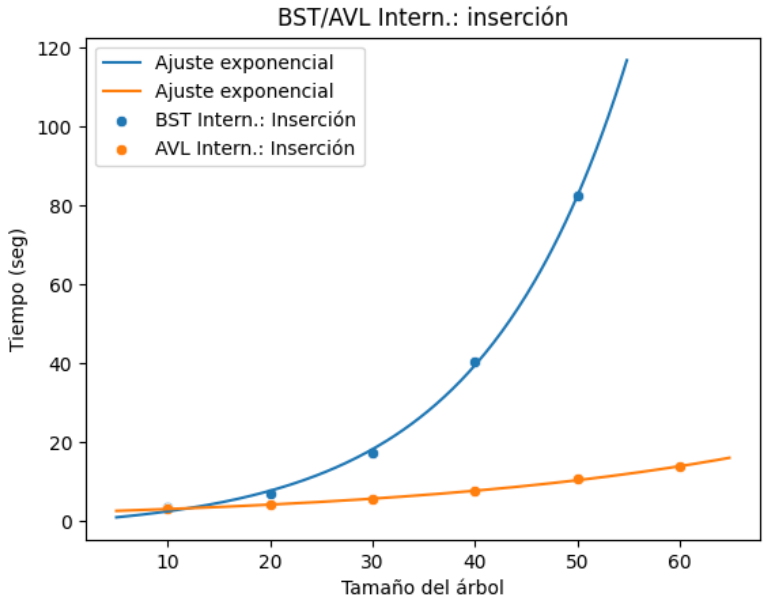


Figura D.16: Tiempos de compilación de la inserción con el enfoque internalista en árboles `BST` y `AVL` de distintos tamaños.

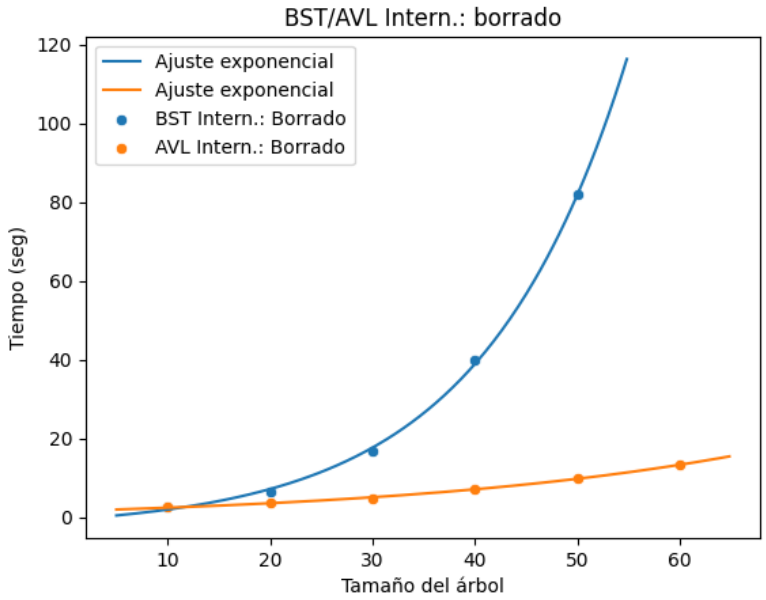


Figura D.17: Tiempos de compilación del borrado con el enfoque internalista en árboles `BST` y `AVL` de distintos tamaños.

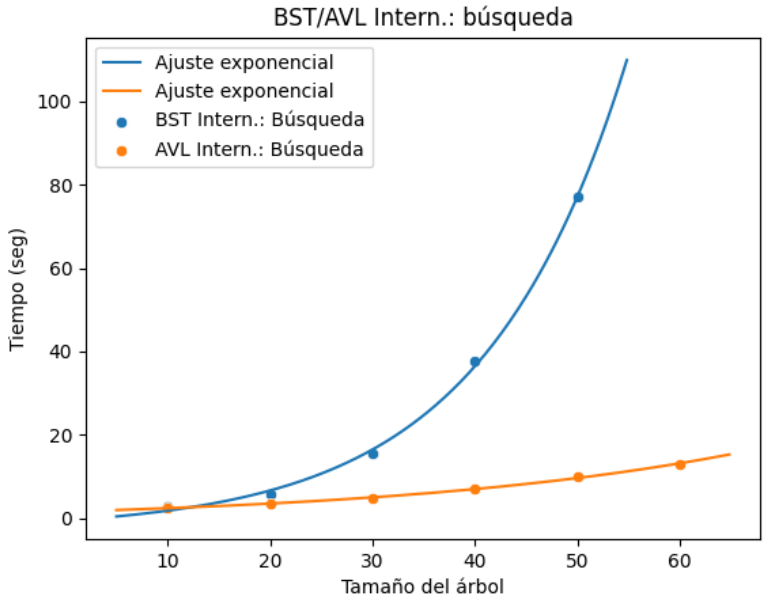


Figura D.18: Tiempos de compilación de la búsqueda con el enfoque internalista en árboles `BST` y `AVL` de distintos tamaños.

Bibliografía

[Chu36] A. Church. "An Unsolvable Problem of Elementary Number Theory". En: *American Journal of Mathematics* 58 (1936), pág. 345.

[Tur36] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction". En: *Proceedings of the London Mathematical Society* s2-43.1 (ene. de 1936), págs. 544-546. ISSN: 0024-6115. DOI: 10.1112/plms/s2-43.6.544. eprint: https://academic.oup.com/plms/article-pdf/s2-43/1/544/4324517/s2-43-6-544.pdf. URL: https://doi.org/10.1112/plms/s2-43.6.544.

[Lev44] Kenneth Levenberg. "A method for the solution of certain non-linear problems in least squares". En: *Quarterly of Applied Mathematics* 2 (1944), págs. 164-168.

[CFC58] H.B. Curry, R. Feys y W. Craig. *Combinatory Logic*. Studies in logic and the foundations of mathematics v. 1. North-Holland, 1958.

[AL62] Georgy Adelson-Velsky y Evgenii Landis. "An algorithm for the organization of information". En: *Proceedings of the USSR Academy of Sciences (in Russian). English translation by Myron J. Ricci in Soviet Mathematics - Doklady* (mar. de 1962), págs. 1259-1263.

[Tai65] W.W. Tait. "Infinitely Long Terms of Transfinite Type". En: *Formal Systems and Recursive Functions*. Ed. por J.N. Crossley y M.A.E. Dummett. Vol. 40. Studies in Logic and the Foundations of Mathematics. Elsevier, 1965, págs. 176-185. DOI: https://doi.org/10.1016/S0049-237X(08)71689-6. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08716896.

[How80] William A Howard. "The Formulae-as-Types Notion of Construction". En: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), págs. 479-490.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201896850.

[SP02] Tim Sheard y Simon Peyton Jones. "Template meta-programming for Haskell". En: *Proceedings of the 2002 Haskell Workshop, Pittsburgh*. Oct. de 2002, págs. 1-16. URL: https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/.

[CH03] James Cheney y Ralf Hinze. *Phantom Types*. 2003.

[KLS04] Oleg Kiselyov, Ralf Lämmel y Keean Schupke. "Strongly typed heterogeneous collections". En: *Proceedings of the ACM SIGPLAN 2004 Haskell Workshop, Haskell'04* (ene. de 2004), págs. 96-107. DOI: 10.1145/1017472.1017488.

[Sul+06] Martin Sulzmann y col. "Understanding functional dependencies via constraint handling rules". To appear in the Journal of Functional Programming. Ene. de 2006. URL: https://www.microsoft.com/en-us/research/publication/understanding-functional-dependencies-via-constraint-handling-rules/.

[Sch+08] Tom Schrijvers y col. "Type Checking with Open Type Functions". En: *ICFP 2008*. Submitted to ICFP'08. Abr. de 2008. URL: https://www.microsoft.com/en-us/research/publication/type-checking-with-open-type-functions/.

[EW12] Richard A Eisenberg y Stephanie Weirich. "Dependently typed programming with singletons". En: *ACM SIGPLAN Notices* 47.12 (2012), págs. 117-130.

[MP12] Simon Marlow y Simon Peyton Jones. "The Glasgow Haskell Compiler". En: *The Architecture of Open Source Applications, Volume 2*. The Architecture of Open Source Applications, Volume 2. Lulu, ene. de 2012. URL: https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/.

[Yor+12] Brent A. Yorgey y col. "Giving Haskell a Promotion". En: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '12. Philadelphia, Pennsylvania, USA: ACM, 2012, págs. 53-66. ISBN: 978-1-4503-1120-5. DOI: 10.1145/2103786.2103795. URL: http://doi.acm.org/10.1145/2103786.2103795.

[LM13] Sam Lindley y Conor McBride. "Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming". En: *SIGPLAN Not.* 48.12 (sep. de 2013), págs. 81-92. ISSN: 0362-1340. DOI: 10.1145/2578854.2503786. URL: http://doi.acm.org/10.1145/2578854.2503786.

[Wad15] Philip Wadler. "Propositions as Types". En: *Commun. ACM* 58.12 (nov. de 2015), págs. 75-84. ISSN: 0001-0782. DOI: 10.1145/2699407. URL: https://doi.org/10.1145/2699407.

[KLS18] Oleg Kiselyov, Ralf Laemmel y Keean Schupke. *HList: Heterogeneous lists*. Hackage. Ver. 0.5.0.0. Feb. de 2018. URL: https://hackage.haskell.org/package/HList (visitado 11-05-2021). En línea.

[20a] *Glasgow Haskell Compiler User's Guide*. 6.4.10. Datatype promotion. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/data_kinds.html (visitado 11-05-2021). En línea.

[20b] *Glasgow Haskell Compiler User's Guide*. 6.4.13. Type-Level Literals. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/type_literals.html#type-level-literals (visitado 11-05-2021). En línea.

[20c] *Glasgow Haskell Compiler User's Guide*. 6.4.8. Generalised Algebraic Data Types (GADTs). GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/gadt.html (visitado 11-05-2021). En línea.

[20d] *Glasgow Haskell Compiler User's Guide*. 6.4.9. Type families. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/type_families.html (visitado 18-06-2021). En línea.

[20e] *Glasgow Haskell Compiler User's Guide*. 6.4.9.2.6. Decidability of type synonym instances. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/type_families.html#decidability-of-type-synonym-instances (visitado 18-06-2021). En línea.

[20f] *Glasgow Haskell Compiler User's Guide*. 6.8.8. Instance declarations and resolution. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/instances.html (visitado 18-06-2021). En línea.

[20g] *Glasgow Haskell Compiler User's Guide*. 6.8.1. Multi-parameter type classes. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/multi_param_type_classes.html#extension-MultiParamTypeClasses (visitado 18-06-2021). En línea.

[20h] *Glasgow Haskell Compiler User's Guide*. 6.11.5.5. Class and instance declarations. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/scoped_type_variables.html#cls-inst-scoped-tyvars (visitado 18-06-2021). En línea.

[20i] *Glasgow Haskell Compiler User's Guide*. 6.11.5. Lexically scoped type variables. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/scoped_type_variables.html (visitado 18-06-2021). En línea.

[20j] *Glasgow Haskell Compiler User's Guide*. 6.8.8.1. Relaxed rules for the instance head. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/instances.html#extension-FlexibleInstances (visitado 18-06-2021). En línea.

[20k] *Glasgow Haskell Compiler User's Guide*. 6.8.8.4. Instance termination rules. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/instances.html#extension-UndecidableInstances (visitado 18-06-2021). En línea.

[20l] *Glasgow Haskell Compiler User's Guide*. 6.13. Template Haskell. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/template_haskell.html (visitado 14-05-2021). En línea.

[20m] *Glasgow Haskell Compiler User's Guide*. 6.4.11. Kind polymorphism. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/poly_kinds.html (visitado 14-05-2021). En línea.

[20n] *Glasgow Haskell Compiler User's Guide*. 6.4.9.2.7. Reducing type family applications. GHC Team. 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/type_families.html#reducing-type-family-applications (visitado 18-06-2021). En línea.

[20o] *Glasgow Haskell Compiler User's Guide*. 6.11.4.4. Pattern type signatures. GHC Team, 2020. URL: https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/scoped_type_variables.html#pattern-type-sigs (visitado 18-06-2021). En línea.

[Mar20a] Simon Marlow, ed. *The Haskell 2010 Language Report*. Chapter 4: Declarations and Bindings. Jul. de 2020. URL: https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-630004.1. (visitado 09-05-2021). En línea.

[Mar20b] Simon Marlow, ed. *The Haskell 2010 Language Report*. Chapter 4: 4.3 Type Classes and Overloading. Jul. de 2020. URL: https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-750004.3 (visitado 18-06-2021). En línea.

[Mar20c] Simon Marlow, ed. *The Haskell 2010 Language Report*. Chapter 3: 3.17 Pattern Matching. Jul. de 2020. URL: https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-580003.17 (visitado 18-06-2021). En línea.

[com21] The SciPy community. *SciPy: Optimization*. curve_fit. Sphinx. Ver. 1.7.1. Mar. de 2021. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html (visitado 19-06-2021). En línea.

[ES21] Richard Eisenberg y Jan Stolarek. *singletons: Basic singleton types and definitions*. Hackage. Ver. 3.0. Mar. de 2021. URL: https://hackage.haskell.org/package/singletons (visitado 11-05-2021). En línea.

[lib21a] libraries@haskell.org. *base: Basic libraries*. GHC.TypeNats. Hackage. Ver. 4.15.0.0. Mar. de 2021. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/GHC-TypeNats.html (visitado 11-05-2021). En línea.

[lib21b] libraries@haskell.org. *base: Basic libraries*. Data.Proxy. Hackage. Ver. 4.15.0.0. Mar. de 2021. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-Proxy.html (visitado 14-06-2021). En línea.

[lib21c] libraries@haskell.org. *base: Basic libraries*. Data.Type.Equality. Hackage. Ver. 4.15.0.0. Mar. de 2021. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-Type-Equality.html (visitado 14-06-2021). En línea.

[lib21d] libraries@haskell.org. *base: Basic libraries*. Hackage. Ver. 4.15.0.0. Mar. de 2021. URL: https://hackage.haskell.org/package/base (visitado 15-06-2021). En línea.

[lib21e] libraries@haskell.org. *base: Basic libraries*. Data.Type.Equality: The equality types. Hackage. Ver. 4.15.0.0. Mar. de 2021. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-Type-Equality.html#g:1 (visitado 16-06-2021). En línea.

[lib21f] libraries@haskell.org. *base: Basic libraries*. Data.Type.Equality: castWith. Hackage. Ver. 4.15.0.0. Mar. de 2021. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-Type-Equality.html#v:castWith (visitado 18-06-2021). En línea.

[lib21g] libraries@haskell.org. *base: Basic libraries*. Data.Type.Equality: gcastWith. Hackage. Ver. 4.15.0.0. Mar. de 2021. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-Type-Equality.html#v:gcastWith (visitado 18-06-2021). En línea.

[lib21h] libraries@haskell.org. *ghc-prim: GHC primitives*. Equality and ordering. Hackage. Ver. 0.7.0. Mar. de 2021. URL: https://hackage.haskell.org/package/ghc-prim-0.7.0/docs/GHC-Classes.html#g:2 (visitado 19-06-2021). En línea.

[lib21i] libraries@haskell.org. *GHC.TypeLits*. User-defined type errors. Hackage. Ver. 0.7.0. Mar. de 2021. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/GHC-TypeLits.html#g:4 (visitado 19-06-2021). En línea.

[21] *The Glasgow Haskell Compiler*. Mar. de 2021. URL: https://www.haskell.org/ghc/ (visitado 11-05-2021). En línea.