Trabajo de Grado

Un Enfoque Ecléctico para la Implementación de Estructuras de Datos

Pablo J. Pedemonte

Directores
Prof. Gabriel Baum Msc. Pablo E. Martínez López

Agradecimientos

En primer lugar quisiera mencionar a mi familia en Rosario: Luis, Walter, Silvia, Ángela y Carlos, por su apoyo continuo aún luego de mi primer año en La Plata, en el que nada fue fácil.

Estoy completamente en deuda con el Bocha y Cabeza, destacados miembros de la "legión Rosarina", a la que también tengo el honor de pertenecer. Su amistad fue siempre ayuda psicológica para llevar adelante mi carrera.

Agradezco también al Laboratorio de Investigación y Formación en Informática Avanzada (LIFIA), del que formo parte desde 1996, por proveer un entorno adecuado para la investigación y el autoaprendizaje. Hay muchos miembros y ex–integrantes del laboratorio que deseo mencionar, desde ya pido disculpas por cualquier omisión. En primer lugar, quiero agradecer a Pablo E. Martínez López (Fidel) y a Gabriel Baum por sus enseñanzas y buenos consejos, tanto para esta tesis como a lo largo de mi licenciatura. Agradezco muy especialmente al resto de los que son o fueron parte del staff del LIFIA–λ: Hernán, Esteban, Eduardo, el Colo, Walter, Daniel y Germán, por su ayuda continua e incondicional, y por haber contribuido al excelente clima que siempre reinó en ese laboratorio. Imposible no mencionar a la "cofradía de la cocina": Sansa, Fedara, Simo, Ramiro, el Guara, Shaka y el Ruso. Agradezco también a Gustavo Nestares, Mariano Barcia, Federico Balaguer, Pálin, Francisco, el Casco, Pipa y Marcelo Tondato.

Quiero agradecer a Guido Macchi, acaso uno de los gurúes de Unix y C más grande que conocí, por compartir siempre su saber y por haber sugerido a la UNLP como una opción posible para mis estudios en informática.

Finalmente, el agradecimiento más especial de todos: a Wanda, por su amor incondicional en todo aspecto, sin el que nada tendría sentido. A Wanda entonces, por ser la otra mitad.

Pablo J. Pedemonte
La Plata, Agosto de 2001

Este documento fue procesado usando el sistema de macros LaTeX 2ε, y el modo Haskell desarrollado por Manuel M. T. Chakravarty (GNU public license, 1998).

Índice General

Capítulo 1

Introducción

Las estructuras de datos son un tema de estudio fundamental para las Ciencias de la Computación. Basta notar que la eficiencia tanto temporal como espacial de un algoritmo depende en gran medida de la calidad de las estructuras de datos que utilice. Además, las estructuras de datos se relacionan con conceptos igualmente importantes como complejidad u ocultamiento de la información, y no se limitan a un dominio específico. Por el contrario, su estudio es de utilidad en diversas área de investigación, como por ejemplo sistemas operativos, compiladores, bases de datos, redes, sistemas de tiempo real, etc.

Este trabajo de grado pone énfasis en el estudio de las estructuras de datos desde el punto de vista del paradigma de programación funcional [Bir98, Dav92, Tho96]. Bajo el paradigma funcional, las estructuras de datos pueden clasificarse de dos maneras, según su implementación se base o no en efectos laterales. Las estructuras de datos que no basan su implementación en efectos laterales se denominan *funcionales puras*; un ejemplo clásico son los árboles o las colas de prioridad. Aquellas que necesitan recurrir a los efectos laterales con el fin de obtener una implementación eficiente se conocen como estructuras de datos *procedurales* o *imperativas* (p. ej., tablas *hash*).

Existen diversos motivos por los cuales el paradigma funcional provee un entorno adecuado para el estudio y desarrollo de estructuras de datos. Los lenguajes funcionales puros carecen de efectos laterales, lo que significa que una expresión puede reemplazarse por cualquier otra que produzca el mismo valor sin cambiar el resultado de un programa. Esto hace que sea sencillo derivar programas funcionales puros o demostrar su correctitud. A su vez, los lenguajes funcionales poseen diversas características de utilidad para el desarrollo de estructuras de datos, tales como alto orden, polimorfismo, chequeo de tipos, y *lazy evaluation* (la propiedad por la que una expresión se computa sólo si es necesario, y en tal caso, sólo una vez). Finalmente, hace

algunos años Wadler introdujo las mónadas [Wad92a, Wad95] y los tipos lineales [Wad93, Wad90] como mecanismos para modelar efectos laterales en los lenguajes funcionales puros sin perder las propiedades de transparencia referencial. Esto hace posible implementar estructuras de datos imperativas.

En base a las motivaciones descriptas anteriormente, esta tesis persigue los siguientes objetivos:

- Implementar dentro del paradigma funcional estructuras de datos tanto funcionales puras como imperativas que no sean triviales y que además utilicen adecuadamente las ventajas que brinda dicho paradigma. Las implementaciones funcionales puras deben ser simples, de modo que sea sencillo razonar sobre su correctitud. Las procedurales deben estar implementadas de modo tal que no sean una mera copia de sus contrapartes imperativas, sino que utilicen las ventajas ofrecidas por el paradigma, comúnmente asociadas a los programas funcionales puros.

- Explorar distintas alternativas de implementación. Se verá que distintos lenguajes funcionales presentan diferentes características que impactan sobre las técnicas de implementación de estructuras de datos. Este trabajo intenta analizar las ventajas y desventajas que ellas presentan.

- Obtener implementaciones eficientes, o sea, con tiempos de ejecución comparables a los obtenidos por una implementación imperativa. Para esto es importante disponer de versiones imperativas con el fin de tener un patrón contra el cual comparar el rendimiento de las implementaciones funcionales.

Debido a que el estudio de las estructuras de datos funcionales enfatiza aspectos diferentes según se traten de funcionales puras o imperativas, este trabajo efectúa tal estudio por separado. Los Cap. 2 y 3 estudian dos estructuras funcionales puras: árboles AVL [PMP95] y *treaps* [AS89], útiles para modelar eficientemente conjuntos ordenados. El Cap. 4 estudia un tipo particular de estructura de datos imperativa llamada árbol de *van Emde Boas* [vEBKZ77], utilizada para implementar colas de prioridad. Un árbol de van Emde Boas modela un subconjunto de un universo finito U, con $|\,U\,| = u$, cuyos elementos soportan aritmética entera. Se trata de una estructura de datos altamente eficiente que implementa las operaciones *insert*, *delete*, *succ* y *pred* en tiempo $O(\log\log(u))$.

Los árboles de van Emde Boas son una estructura imperativa con una implementación compleja, por lo que primero se estudia una especificación en pseudo–código de sus operaciones. En cambio, los árboles AVL y los

treaps son estructuras de datos funcionales puras cuyas operaciones se pueden especificar directamente mediante un lenguaje funcional. Esta es una característica importante del paradigma funcional puro: en general, las implementaciones funcionales puras tienden a ser sencillas y sirven también como especificación del problema que están resolviendo.

Con el fin de poder explorar diferentes alternativas de implementación, se utilizaron dos lenguajes funcionales puros: Haskell [PJH99] y Clean [PvE98]. Las versiones imperativas para comparar los resultados fueron implementadas en C [KR88]. Además, las estructuras implementadas fueron sometidas a una serie de casos de prueba, con la intención de obtener datos que permitan analizar y comparar su eficiencia. Esto hace posible determinar qué porciones de la codificación deben ser optimizadas y cómo debe llevarse a cabo dicha optimización.

Este capítulo está organizado de la siguiente manera. La Secc. 1.1 describe los tres lenguajes utilizados a lo largo de este trabajo. Las secciones 1.2 y 1.3 discuten, respectivamente, las estructuras de datos puras y procedurales desde el punto de vista de la programación funcional. En la Secc. 1.4 se describen brevemente las pruebas efectuadas para el análisis de eficiencia y los resultados obtenidos. Finalmente, la Secc. 1.5 analiza las alternativas con las que es posible continuar el trabajo realizado.

En el Ap. A se da una breve introducción a los conceptos básicos de la programación funcional más relevantes para el desarrollo de esta tesis, con el fin de servir de guía al lector que precise refrescar sus conocimientos sobre el tema.

1.1 Haskell, Clean y C

Para implementar las estructuras de datos descriptas se utilizaron los lenguajes Haskell, Clean y C. Todos los programas fueron desarrollados bajo Linux, debido a que es una de las plataformas para la cual existen versiones eficientes de estos tres lenguajes.

Haskell

Haskell es el lenguaje funcional *lazy standard*. Un lenguaje *lazy* sigue la regla de "computar si es necesario, y en tal caso, sólo una vez". Entre otras ventajas, la *lazy evaluation* disminuye el grado de cohesión entre funciones [Hug89], y permite el uso de estructuras infinitas (ver el Ap. A). Otra característica importante de los lenguajes funcionales puros es el manejo automático de memoria. La memoria se libera automáticamente (cuando el espacio disponible

disminuye por debajo de un cierto umbral) mediante un mecanismo llamado *garbage collection*, que consiste en identificar y liberar las celdas de memoria ya no referenciadas. Es deseable mantener acotado el consumo de memoria, ya que de otro modo las fases de *garbage collection* podrían aumentar significativamente el tiempo total de ejecución.

Además de las características comunes a todos los lenguajes funcionales *lazy*, Haskell provee un operador para forzar que una función sea estricta, o sea, a que evalúe sus argumentos antes de ser ejecutada (alterando así la propiedad de *lazyness*). Por otra parte, Haskell modela computaciones basadas en estado mediante mónadas, una familia particular de TAD's que permite modelar efectos laterales de manera pura, ocultando el estado a ser modificado mediante dichos efectos y evitando así su manipulación explícita. De este modo, las operaciones monádicas (o sea, las del TAD) permiten asegurar que en todo momento el estado es referenciado por a lo sumo una única expresión, por lo que puede ser actualizado de manera imperativa sin introducir un efecto lateral. Haskell provee también herramientas para analizar el rendimiento tanto temporal como espacial de un programa dado (*time* y *heap profilers*, respectivamente [SPJ95]). El compilador Haskell utilizado fue el `ghc` versión 4.08, implementado por la universidad de Glasgow [PJHH+93].

Clean

Clean es un lenguaje funcional diseñado para producir código altamente eficiente. Si bien presenta numerosas características similares a las de Haskell (p. ej. ser *lazy*), este lenguaje fue elegido porque posee dos diferencias importantes. Por un lado, permite denotar computaciones estrictas a nivel de tipos en lugar de recurrir a un operador. Esto otorga una flexibilidad mucho mayor para definir qué partes de un programa deben ser computadas de manera estricta y cuáles no. Además, Clean modela efectos laterales mediante una variante de tipos lineales llamada *unique types* [Hud92, SBvEP94, Kes94]. Una expresión con tipo *unique* sólo puede tener a lo sumo una referencia a lo largo de la ejecución del programa, por lo que puede ser modificada sin introducir efectos laterales. El compilador Clean provee herramientas de *profiling* para plataformas Win32, pero no para Linux. No obstante, mediante el estudio de los programas Clean en Win32, fue posible analizar el rendimiento de las versiones Linux. La versión de Clean utilizada fue la 1.3.3 [PvE98].

Lenguaje C

El lenguaje C es utilizado exclusivamente para tener implementaciones imperativas "óptimas" contra las cuales puedan ser comparadas las versiones

funcionales. Dichas implementaciones son también de utilidad para comparar la complejidad del código imperativo con el de las versiones funcionales. El compilador utilizado fue el `gcc` versión 2.95.2, para `glibc2`.

1.2 Estructuras de Datos Funcionales Puras

Existen diversos tipos de estructuras de datos funcionales puras que pueden ser implementadas con eficiencia razonable mediante técnicas tales como amortización o *lazy evaluation*. Este tópico es tema de investigación con resultados promisorios [Oka95, Oka96a, Oka96b], habiéndose desarrollado bibliotecas que implementan de manera eficiente colas, deques, árboles binomiales, árboles balanceados, etc. [Oka99].

Muchas de estas estructuras presentan implementaciones procedurales mediana o altamente complejas, lo que hace que sean difíciles de comprender y estudiar. Uno de los objetivos de esta tesis es obtener implementaciones sencillas para este tipo de estructuras de datos, y por lo tanto mostrar que esa complejidad no es inherente a la estructura en sí, sino que está provocada por el carácter imperativo de su implementación. Esto es debido a dos factores fundamentales:

Representación interna: las estructuras de datos imperativas basan su representación interna en construcciones procedurales tales como punteros, *variant-records*, etc. Esto provoca que su implementación esté manipulando continuamente punteros y alocando o liberando memoria, lo que oculta la esencia del algoritmo.

Efectos laterales: los efectos laterales (presentados por todos los lenguajes imperativos) dificultan la demostración de la correctitud de las implementaciones. En lugar de razonar algebraicamente sobre las expresiones que conforman el programa, es necesario razonar sobre un sistema formal, que dependiendo de la semántica del lenguaje, puede llevar a demostraciones extensas y complejas.

Al implementar estructuras de datos no procedurales mediante un lenguaje funcional puro, los algoritmos pasan a ser más claros debido a que dejan de estar planteados en términos imperativos. Además, se hace posible demostrar propiedades tales como correctitud o terminación, debido la ausencia de efectos laterales. No obstante, existe un precio a pagar: la eficiencia de los programas funcionales es inferior a las de los imperativos. Esta diferencia se debe principalmente a que el manejo automático de la memoria que realizan los lenguajes funcionales, si bien extremadamente conveniente,

es (hoy día) menos eficiente que el provisto por los lenguajes procedurales. Debido a esto, es necesario recurrir a técnicas de *profiling* para optimizar el uso de memoria (y con ello el tiempo de ejecución) por parte de dichas implementaciones [ML98]. Esta tesis pretende mostrar, sin embargo, que es posible acercarse razonablemente a la eficiencia de los lenguajes imperativos sin sacrificar claridad de código.

1.3 Estructuras de Datos Imperativas

Las estructuras de datos funcionales puras no bastan para el desarrollo de aplicaciones eficientes, ya que en general las operaciones de *update* no pueden ser realizadas en tiempo constante (para modificar un elemento debe retornarse una copia de la estructura de datos, en lugar de sólo modificar dicho elemento). Esto es, una estructura de datos funcional pura efectuará *updates* (en el peor caso) en tiempo proporcional a la cantidad de elementos. Esto no alcanza para implementar adecuadamente técnicas como *hashing*, que tienen como objetivo proveer operaciones con tiempo constante amortizado. Para poder alcanzar este grado de complejidad, es necesario incluir estructuras de datos procedurales.

Estas abstracciones basan su implementación en una operación denominada *update in place*. Esta operación corresponde a la idea imperativa de asignación: reescribir la posición de memoria donde se halla almacenada la variable con el nuevo valor. Si bien esta operación parece trivial, en general no puede ser ejecutada por un lenguaje funcional *lazy* sin correr el riesgo de introducir un efecto lateral; esto es debido a que no es posible determinar a priori que expresiones de un programa serán ejecutadas y en qué orden. Por lo tanto, esta operación sólo puede ser incluída mediante técnicas especiales que aseguran que no se producirá un efecto lateral al ser ejecutada. A tal propósito, Haskell utiliza mónadas, y Clean usa *unique types* (ver la Secc. 1.1). Esto hace que no sea trivial implementar eficientemente estructuras de datos procedurales mediante un lenguaje funcional puro. Un ejemplo clásico lo constituyen los arreglos, los cuales no presentan una implementación funcional sencilla. En consecuencia, los lenguajes funcionales en general no proveen aún bibliotecas o *frameworks* con implementaciones eficientes de estructuras de datos procedurales de uso común; al contrario de lo que sucede con otros lenguajes como C++ [Str94] (que incluye como biblioteca standard a STL [SL95]) o Java 1.2 [AGH01], que incluye en su biblioteca de clases el *Java Collections Framework*.

En este trabajo se intenta mostrar que es posible llenar ese vacío, esto es, implementar eficientemente estructuras de datos procedurales utilizando

para ello un lenguaje funcional puro. La idea consiste en unificar las técnicas utilizadas por dichos lenguajes para introducir efectos laterales junto con sus características originales (*lazy evaluation*, polimorfismo paramétrico, etc.), de modo que las implementaciones obtenidas presenten a la vez la claridad usual de los programas funcionales puros y la eficiencia de los programas imperativos.

A pesar de su importancia, las estructuras de datos procedurales aún no han sido estudiadas lo suficiente por la comunidad funcional. Por lo tanto, dicho tópico puede ser considerado como el aporte principal de esta tesis al estudio de la implementación de estructuras de datos eficientes bajo el paradigma funcional puro.

1.4 Análisis de Eficiencia y Resultados

1.4.1 Análisis de Eficiencia

Para analizar la eficiencia de las estructuras de datos implementadas se recurrió a diferentes casos de prueba. Estos consistieron en insertar 10^3, 10^4 y 10^5 elementos generados pseudo–aleatoriamente, seguidos de una operación con tiempo $O(n)$ para el peor caso: computar la altura de la estructura resultante. Se eligió este caso de prueba porque las operaciones de inserción *crean* nodos, por lo que es posible analizar no sólo el tiempo de ejecución, sino también el consumo de memoria resultante. La obtención de la altura se incluye para forzar la inserción de todos los elementos; debido a que los lenguajes funcionales utilizados son *lazy*, si no lleva a cabo alguna computación sobre la estructura de datos resultante, las inserciones no serán efectuadas.

Los datos provenientes de los casos de prueba fueron obtenidos de diversas maneras, dependiendo del lenguaje estudiado. Para Haskell, se obtuvieron mediante las herramientas de *profiling* provistas por el lenguaje. Si bien la performance de los programas Clean fue medida bajo Linux en base a los reportes generados por dichos programas una vez finalizada su ejecución, estos fueron analizados bajo Windows debido a que Clean todavía no provee *profilers* para Linux. Sin embargo, este estudio hizo posible detectar y optimizar fragmentos de código que provocaban bajo rendimiento, ya que las implementaciones de Clean para ambas plataformas funcionan de manera similar. El compilador C sólo provee un *time profiler*, debido a que en los lenguajes imperativos el análisis del consumo de memoria no es tan importante como lo es para los lenguajes funcionales. No obstante, esto basta para comparar los tiempos de ejecución de las versiones funcionales contra el de las versiones imperativas.

1.4.2 Resultados

Los tiempos de ejecución obtenidos son satisfactorios. El peor tiempo corresponde a la implementación Haskell de árboles de van Emde Boas, y tarda sólo entre 10 y 18.84 veces más, según el tamaño de la entrada. Exceptuando este caso, las demás implementaciones funcionales tardan a lo sumo 5.56 veces más que sus contrapartes en C. Estas proporciones son muy favorables, e incluso comparables con otros lenguajes de uso común, tales como Java y Visual Basic. Esto lleva a concluir que es posible y conveniente implementar implementar estructuras de datos de manera eficiente utilizando lenguajes funcionales puros: las pérdidas de performance se ven compensadas por las ventajas que ofrece el paradigma (por ejemplo, el código de las versiones funcionales es mucho más conciso y fácil de comprender que el de las versiones imperativas). Esto se verifica especialmente para el caso de las estructuras de datos funcionales puras.

1.5 Trabajo Futuro

Hay diversos aspectos de implementación sobre los que se puede continuar el trabajo desarrollado hasta ahora. Por ejemplo, a pesar de que las versiones funcionales han sido optimizadas a lo largo de su desarrollo, todavía existen fragmentos de código que hacen uso ineficiente del heap. Para las implementaciones Haskell se debería investigar también la utilización de directivas de compilación para optimizar el código ejecutable, tales como especialización e *inlining* (o sea, el reemplazo de una llamada a función por su código)

Por otra parte, existe un compilador para un subconjunto de Haskell llamado `nhc` [Röj94], el cual fue desarrollado específicamente para producir programas eficientes en el uso de memoria. En efecto, el consumo de heap está optimizado, y además provee un *heap profiler* de alta calidad. Si bien es un compilador experimental y más lento que el `ghc`, sería de utilidad para estudiar con un mayor nivel de detalle el rendimiento espacial de las estructuras de datos implementadas.

Es importante notar que el estudio de la implementación de estructuras de datos eficientes no basta por sí solo para permitir la construcción de una biblioteca que pueda llegar a ser un componente más del estado del arte de los lenguajes funcionales. Más allá de investigar la implementación de estructuras de datos avanzadas, es necesario estudiar el diseño de tal librería. Las características deseables del paradigma funcional hacen que existan numerosas alternativas a ser consideradas. Entre ellas, se puede mencionar las siguientes: representaciones adaptativas, uso de clases multi-parametrizables,

grado de polimorfismo y tipo de estructuras de datos a implementar. Debido a que estas ideas se centran en el diseño de estructuras de datos en lugar de su implementación, están fuera del alcance de este trabajo y por lo tanto se omite su estudio. El lector interesado puede encontrar en [PJ97] una discusión detallada sobre este tema.

A partir de este trabajo puede notarse que es posible desarrollar en un entorno funcional puro estructuras de datos eficientes, ya sea funcionales puras o procedurales. Este hecho es gran importancia, porque estructuras de datos eficientes permiten codificar programas eficientes. Sin embargo, esta tarea no siempre resulta sencilla. Se requiere un grado de experiencia considerable para obtener un programa funcional puro lo bastante eficiente como para no requerir sesiones de *profiling*. Por lo tanto, es necesario seguir investigando temas tales como técnicas de *profiling* y optimización del uso de memoria, con el objetivo de hacer de los lenguajes funcionales puros herramientas aún más aptas para el desarrollo de estructuras de datos (y aplicaciones en general) eficientes.

Capítulo 2

Árboles AVL

Los árboles AVL son una estructura de datos muy utilizada en las ciencias de la computación. Sin embargo, su implementación no suele ser una tarea sencilla, lo cual es particularmente cierto para las operaciones de inserción y borrado. No obstante, esta complejidad no es inherente a la estructura de datos en sí, sino que puede atribuírse en gran medida a los detalles de implementación que deben ser tenidos en cuenta para desarrollarla.

Este capítulo muestra que las dificultades aparentemente crónicas que presenta la implementación de árboles AVL pueden ser evitadas valiéndose de las propiedades del paradigma funcional. Los lenguajes funcionales puros permiten lograr un alto nivel de abstracción, de forma que el programador pueda razonar matemáticamente sobre la estructura de un programa, obviando detalles de implementación irrelevantes. De este modo es posible obtener una implementación de árboles AVL concisa, basada en la definición formal de árbol AVL bien formado, y además demostrar formalmente la correctitud de dicha implementación.

Podría pensarse que la claridad y correctitud se pagan con eficiencia. Sin embargo, mostraremos que la eficiencia de las implementaciones funcionales de árboles AVL es comparable a la obtenida por una imperativa. Si bien los lenguajes funcionales puros todavía no igualan a los imperativos en eficiencia temporal o espacial, la pérdida de performance de una implementación funcional contra una procedural puede ser muy baja o incluso nula.

Este capítulo está organizado de la siguiente manera. La Secc. 2.1 da la definición de árboles AVL. Las Secc. 2.2 y 2.3 describen la implementación Haskell y Clean de árboles AVL, respectivamente. La Secc. 2.4 estudia una implementación en C. En la Secc. 2.5 se prueba formalmente la correctitud de las implementaciones funcionales. Finalmente, la Secc. 2.6 analiza y compara la eficiencia de las implementaciones provistas, mostrando que la eficiencia de las versiones funcionales es comparable a la obtenida por la versión C.

2.1 Defición de Árboles AVL

Un árbol AVL es un árbol binario de búsqueda que implementa las operaciones usuales de búsqueda, inserción y borrado, y que además cumple con la propiedad de estar balanceado. Esto significa que si no se trata de una hoja, los subárboles correspondientes están balanceados y su diferencia de altura es menor o igual que uno. Para formalizar este requerimiento se definirá un predicado llamado *isBalanced*, que modela la condición de balance que deberá cumplirse para todo árbol AVL t bien formado. Una primera aproximación a su definición consiste es separar la condición de balance en las siguientes alternativas:

- Si t es un árbol formado por una única hoja, entonces está balanceado.
- En caso contrario, está balanceado si sus subárboles lo están y la diferencia de entre ellos altura no es mayor a uno.

Para expresar la formalización de estas dos alternativas se utiliza la siguiente notación: para todo árbol binario t, d_t denota su profundidad, $t.l$ su subárbol izquierdo, y $t.r$ el subárbol derecho.

$$\begin{aligned} &\mathit{isBalanced}(t) \equiv \\ &\quad d_t = 0 \ \vee \ (\mathit{isBalanced}(t.l) \wedge \mathit{isBalanced}(t.r) \wedge |\, d_{t.l} - d_{t.r} \,| \leqslant 1) \end{aligned}$$

Una implementación correcta de árboles AVL debe asegurar la invariancia de este predicado para todo árbol AVL válido. La Secc. 2.5 demuestra que las implementaciones Haskell y Clean cumplen con esta condición.

2.2 Árboles AVL: Versión Haskell

La versión Haskell de árboles AVL está compuesta por dos clases de funciones. Por un lado, se tienen las funciones que implementan la funcionalidad básica de esta estructura de datos: las operaciones de búsqueda, inserción y borrado. A su vez, estas operaciones se basan en funciones auxiliares que implementan el algoritmo de balance, las cuales aseguran la invariancia del predicado *isBalanced*, manteniendo a su vez la condición de árbol de búsqueda. El estudio de la versión Haskell se llevará a cabo en este mismo orden: se discutirá la implementación de las operaciones básicas, y luego la del algoritmo de balance.

2.2.1 Operaciones

La implementación Haskell estará basada en pares *clave-valor*. Para efectuar una inserción hay que pasar como parámetros un valor de tipo v y su clave asociada, de tipo k. Para verificar si un elemento pertenece al árbol o para borrarlo, sólo se requiere como parámetro su clave. El tipo *Maybe v* permite modelar una computación que puede fallar de manera controlada, o en caso de tener éxito retornar un valor de tipo v (ver el Ap. A). Las operaciones principales provistas para árboles AVL son:

```
emptyAVL    :: Ord k => Tree k v
isEmptyAVL  :: Ord k => Tree k v -> Bool
searchAVL   :: Ord k => Tree k v -> k -> Maybe v
insertAVL   :: Ord k => Tree k v -> k -> v -> Tree k v
deleteAVL   :: Ord k => Tree k v -> k -> Tree k v
```

El tipo k de las claves debe ser tal que sea posible comparar sus elementos mediante una relación de orden. En Haskell, este requerimiento se expresa mediante el contexto *Ord k*.

2.2.2 Estructura Interna

Un árbol puede tomar dos formas diferentes: puede estar vacío o tener profundidad mayor que cero, en cuyo caso contendrá una clave con su valor asociado y dos subárboles, posiblemente vacíos. Si bien esta definición es correcta, no es conveniente debido a que la implementación requiere la profundidad del árbol para poder mantener el balance mediante la reorganización de su estructura. Como el cómputo de la profundidad de un árbol requiere visitar cada subárbol, el costo de calcularla para un árbol con n nodos es $O(n)$. La solución consiste en agregar a cada nodo un atributo extra que indicará la profundidad del subárbol cuya raíz es dicho nodo. Este valor se calcula durante la construcción del árbol, de manera que su costo se distribuye de manera uniforme entre las distintas operaciones de inserción. Si bien la obtención de la profundidad de un subárbol puede hacerse ahora en $O(1)$, esto implica agregar n celdas de memoria (o sea, la constante de la complejidad espacial aumenta en 1). Este es un ejemplo del *trade–off* que existe entre complejidad temporal y espacial.

La estructura interna final del árbol se lista a continuación. También se define la función auxiliar *depth*, que será de utilidad en lo que resta de la

implementación. Esta función retorna la profundidad de un árbol t si no está vacío, o cero en caso contrario (o sea, $depth\ t = d_t$).

```
data Tree k v = Empty | Node Int (Tree k v) k v (Treek v)

depth :: Tree k v → Int
depth Empty = 0
depth (Node d _ _ _ _ ) = d
```

2.2.3 Búsqueda, Inserción, Borrado

La operación de búsqueda no altera el balance del árbol, lo que hace que su implementación sea trivial e igual a la de árboles binarios de búsqueda ordinarios, por lo que será omitida.

La inserción se define de la siguiente manera: si el árbol donde será insertado el nuevo par clave-valor está vacío, se debe crear una hoja; en caso contrario se debe insertar el par en el subárbol izquierdo o derecho, según el valor de la clave de modo que se mantenga la condición de árbol de búsqueda. Lo que diferencia a esta operación de su símil en árboles de búsqueda ordinarios es que es necesario asegurar el invariante *isBalanced*, de lo cual se encarga la función *join*. Esta función compensa la diferencia de altura entre los árboles recibidos, reorganizando sus subárboles de modo tal que la condición de búsqueda sea mantenida y que la diferencia de altura entre los componentes del árbol resultante pase a ser menor que dos.

```
insertAVL :: Ord k ⇒ Tree k v → k → v → Tree k v
insertAVL Empty k_n v_n = Node 1 Empty k_n v_n Empty
insertAVL (Node d l k v r) k_n v_n =
case (compare k_n k) of
  EQ → Node d l k v_n r
  LT → join (insertAVL l k_n v_n) k v r
  GT → join l k v (insertAVL r k_n v_n)
```

El borrado es una operación levemente más compleja. Al igual que su implementación en árboles no balanceados, la operación debe ser invocada recursivamente en el subárbol donde está el elemento a borrar. Como la operación

puede alterar el balance, nuevamente hay que recurrir a la función *join* para mantenerlo. Esta porción del algoritmo está implementada por la función *deleteAVL*. Una vez encontrado el nodo que debe ser borrado, se tienen las alternativas usuales:

- Si el subárbol izquierdo está vacío, retornar el derecho. Dualmente, si el derecho es vacío, retornar el izquierdo.
- En caso contrario, eliminar el predecesor inorder del nodo a borrar y reemplazar su par clave-valor por el predecesor obtenido. La eliminación del predecesor inorder puede desbalancear el árbol, por lo que hay que invocar a la función *join* para mantenerlo.

Estos dos casos son implementados por la función *delete′*. Las dos primeras ecuaciones de *delete′* corresponden al primer caso, o sea, cuando alguno de los subárboles es vacío. El segundo está implementado por la última ecuación, la cual invoca a la función *tspan*. Esta función elimina del subárbol izquierdo el predecesor inorder del nodo a borrar, retornando el nuevo subárbol izquierdo y el par clave-valor correspondiente a dicho predecesor. Ambos son combinados con el subárbol derecho original, obteniéndose así el árbol buscado. El código completo de las tres funciones que implementan el algoritmo de borrado es el siguiente:

```
deleteAVL :: Ord k ⇒ Tree k v → k → Tree k v
deleteAVL Empty _ = Empty
deleteAVL t@(Node d l k v r) k_d =
 case (compare k_d k) of
   LT → join (deleteAVL l k_d) k v r
   GT → join l k v (deleteAVL r k_d)
   EQ → delete′ l r

delete′ :: Tree k v → Tree k v → Tree k v
delete′ Empty r = r
delete′ l Empty = l
delete′ l r     = let(l′, k′, v′) = tspan l
                  in join l′ k′ v′ r

tspan :: Tree k v → (Tree k v, k, v)
tspan (Node _ l k v Empty) = (l, k, v)
tspan (Node _ l k v r) = let(r′, k′, v′) = tspan r
                         in (join l k v r′, k′, v′)
```

2.2.4 La Función *join*

Una vez ejecutada una inserción o eliminado un nodo, el árbol resultante puede no satisfacer el predicado *isBalanced*. La función *join* asegura su invariancia a la vez que mantiene la condición de búsqueda, produciendo un árbol AVL a partir de los dos árboles balanceados y el par clave-valor recibidos. Para esto, *join* opera en función de la diferencia de profundidad de los dos árboles a unir:

```
join :: Tree k v → k → v → Tree k v → Tree k v
join l k v r
  | abs (d_l − d_r) ⩽ 1 = sJoin l k v r
  | d_l == d_r + 2      = lJoin l k v r
  | d_l + 2 == d_r      = rJoin l k v r
  | d_l > d_r + 2       = join ll lk lv (join lr k v r)
  | d_l + 2 < d_r       = join (join l k v rl) rk rv rr
  where
    d_l                 = depth l
    d_r                 = depth r
    (Node _ ll lk lv lr) = l
    (Node _ rl rk rv rr) = r
```

Si la diferencia de profundidad entre los dos árboles es menor o igual que uno, entonces su pueden unir directamente sin que se pierda el balance. La función *sjoin* lleva a cabo esta operación, a la vez que calcula la altura del árbol creado:

```
sJoin :: Tree k v → k → v → Tree k v → Tree k v
sJoin l k v r = Node (1 + max (depth l) (depth r)) l k v r
```

Si la diferencia de profundidad es exactamente dos, se invoca a las funciones *ljoin* o *rjoin*, dependiendo de si el árbol izquierdo es más profundo que el derecho o viceversa, respectivamente. Para los dos casos en que la diferencia de altura es mayor que dos, se computan dos llamadas anidadas a *join*. Notar que la invocación interior produce un árbol cuya altura es mayor que la del menos profundo, ya que para el caso $d_l > d_r + 2$, se verifica que $d_{lr} > d_r$ y

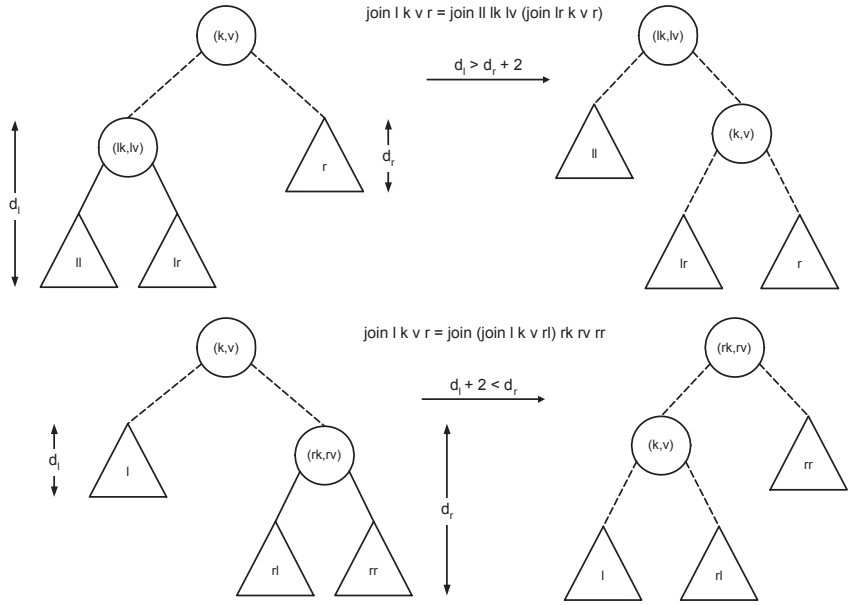

Figura 2.1: Función *join*: caso recursivo

para el caso $d_l + 2 < d_r$ se cumple que $d_l < d_{rl}$. De este modo, se compensan las diferencias de altura iniciales, y la invocación exterior a *join* produce un árbol balanceado. La Fig. 2.1 ilustra este proceso. Las líneas punteadas que unen un nodo con dos subárboles denotan la invocación a *join* con los subárboles y el nodo mencionados como parámetros. Notar que el resultado es también un árbol de búsqueda.

Este algoritmo así codificado funciona sólo con un lenguaje de programación *lazy*, ya que la función *join* asume en la cláusula *where* que los árboles recibidos no son vacíos. Pero como las expresiones donde esto es asumido se evalúan únicamente cuando la profundidad es mayor que uno, por *laziness* el algoritmo es correcto. Sin embargo, puede modificarse fácilmente para que tenga en cuenta los casos vacíos incluso en un lenguaje estricto, aunque incrementando levemente su complejidad.

2.2.5 Las funciones *ljoin* y *rjoin*

Estas funciones son invocadas cuando la diferencia de altura entre los árboles a unir es igual a dos. La función *ljoin* se invoca cuando el árbol izquierdo es dos niveles más profundo que el derecho; en caso contrario, se invoca a *rjoin*. Al igual que *join*, estas funciones compensan la diferencia de altura entre los árboles recibidos reorganizando sus subárboles de modo tal que la

condición de búsqueda sea mantenida y que la diferencia de altura entre los componentes del árbol resultante pase a ser menor que dos. La función *ljoin* puede implementarse así:

```
lJoin :: Tree k v → k → v → Tree k v → Tree k v
lJoin l k v r
  | d_ll ⩾ d_lr    = sJoin ll lk lv (sJoin lr k v r)
  | otherwise = sJoin (sJoin ll lk lv lrl)
                      lrk lrv
                      (sJoin lrr k v r)
    where
      (Node _ ll lk lv lr)      = l
      d_ll                      = depth ll
      d_lr                      = depth lr
      (Node _ lrl lrk lrv lrr)  = lr
```

La reorganización de nodos y subárboles efectuada por dicha función se ilustra en la Fig. 2.2. El efecto de la función *rjoin* es dual, por lo que se omite la figura correspondiente y su código.

Notar que ambas funciones (y por lo tanto, todas las que componen el algoritmo de balance) preservan la condición de búsqueda, dado que después de ser aplicadas todo subárbol continúa estando a la derecha o a la izquierda de su nodo padre original.

Un aspecto importante a tener en cuenta es la eficiencia espacial de este algoritmo. Dado que las funciones de balance simplemente reorganizan el árbol, no deberían crear nuevos nodos. Sin embargo, las funciones de balance ilustradas en las figuras 2.1 y 2.2 *crean* nuevos nodos al ser aplicadas, en lugar de actualizar los ya existentes. Para ver por qué, considérese el siguiente ejemplo:

```
f = let t_1 = join t  ...
        t_2 = join t_1 ...
    in ... t_1 ... t_2 ... t_1 ...
```

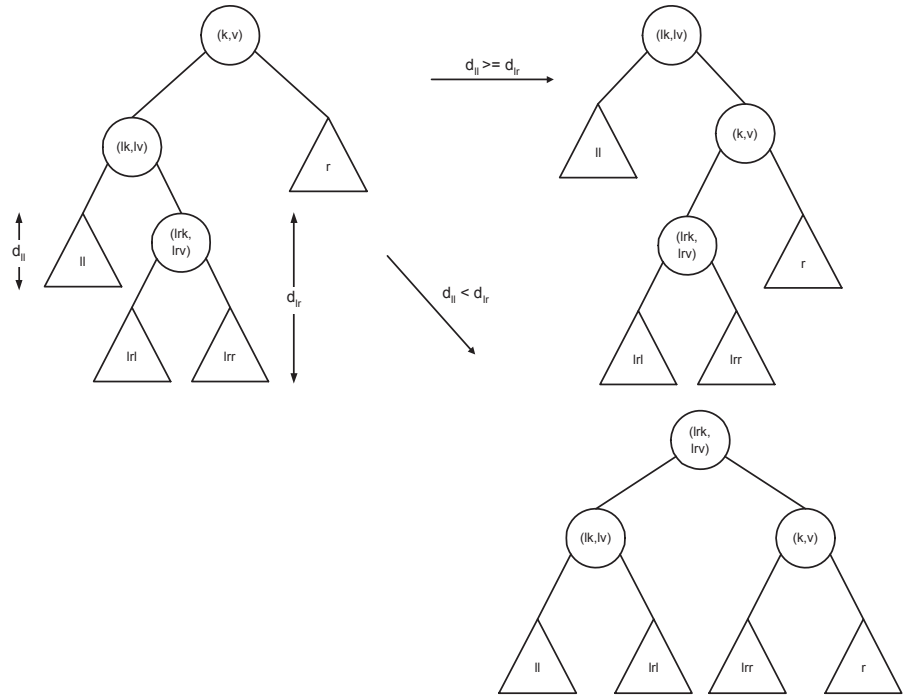

Figura 2.2: Función *ljoin*

Si t_1 y t_2 fuesen resultado de actualizar en lugar de copiar t y t_1 respectivamente, la segunda referencia a t_1 encontraría un árbol distinto al t_1 de la primera; esto es, se habría introducido un efecto lateral. Esta situación por la cual es necesario copiar una estructura en lugar de actualizarla se conoce como el problema del *update in place* (discutido en el Ap. A), y es una de las causas principales de la diferencia de performance de los lenguajes funcionales puros con respecto a los imperativos. En efecto, en la Secc. 2.4 se verá que mediante el uso apropiado de punteros, la versión C no necesita crear nuevos nodos para balancear el árbol.

2.3 Árboles AVL: Versión Clean

La diferencias que muestra esta versión con respecto a la implementación Haskell vienen dadas por la capacidad de Clean para incluir explícitamente anotaciones de *strictness* en el código fuente mediante el operador (!). Estas anotaciones permiten declarar como estrictas ciertas porciones del código o determinadas subestructuras del tipo algebraico que define a los árboles AVL. Al hacer estrictos todos estos componentes de la estructura de datos se

evita la proliferación en el heap de nodos representando computaciones *lazy* a ser evaluadas (*closures*), lo cual provoca una disminución en el consumo de memoria. En base a la anotaciones de *strictness*, el tipo algebraico que define a los árboles AVL fue modificado de la siguiente manera:

$$:: \mathit{Tree}\ k\ v\ =\ \mathit{Empty}\ \mid\ \mathit{Node}\ !\mathit{Int}\ !(\mathit{Tree}\ k\ v)\ !k\ v\ !(\mathit{Tree}\ k\ v)$$

Para todo árbol AVL pasan a ser estrictos su profundidad, ambos subárboles y la clave de su nodo raíz. El valor asociado permanece *lazy*, de modo que un árbol AVL pueda almacenar expresiones que requieren de la *lazy evaluation* para que un programa no falle (por ejemplo, estructuras infinitas). La propiedad de Clean de declarar código como estricto también fue utilizada en la signatura de las operaciones provistas, que pasaron a ser las siguientes:

$$\begin{array}{lll}
\mathit{emptyAVL} & :: \mathit{Tree}\ k\ v & \\
\mathit{isEmptyAVL} & :: !(\mathit{Tree}\ k\ v)\ \rightarrow\ \mathit{Bool} & \\
\mathit{searchAVL} & :: !(\mathit{Tree}\ k\ v)\ !k\ \rightarrow\ \mathit{Maybe}\ v & \mid <, ==\ k \\
\mathit{insertAVL} & :: !(\mathit{Tree}\ k\ v)\ !k\ \ v\ \rightarrow\ \mathit{Tree}\ k\ v & \mid <, ==\ k \\
\mathit{deleteAVL} & :: !(\mathit{Tree}\ k\ v)\ !k\ \rightarrow\ \mathit{Tree}\ k\ v & \mid <, ==\ k
\end{array}$$

Al igual que con la implementación Haskell, el tipo k debe ser comparable. Estas restricciones se expresan en Clean mediante la notación $(|<, ==\ k)$. Para todas las operaciones, se declaran estrictos el árbol AVL y la clave pasada como parámetro. También pasaron a ser estrictos todos los fragmentos de código que declaran expresiones locales, salvo los *where lazy* de la función *join* (Secc. 2.2.4). Exceptuando esta función, fueron eliminadas las sentencias *where* restantes, y los *let*'s fueron reemplazados por *let* estrictos (*let*!). Si bien esto podría suponer trabajo extra dado que ahora se computan expresiones que por *laziness* podrían no ser evaluadas, para esta implementación en particular todas las expresiones declaradas en los *let*'s son computadas siempre. De este modo, el efecto de hacer estrictas las sentencias *let* no introduce un costo adicional. Estas modificaciones también suponen una disminución del consumo de heap por parte de la implementación Clean, dado que una vez más se evita la creación de *closures* que hagan referencia a las expresiones

locales a ser evaluadas. Como estas expresiones están declaradas en un contexto estricto, son evaluadas antes del cuerpo del *let* donde están declaradas y sólo se almacena una referencia al resultado de dicha computación.

Aunque las modificaciones descriptas son menores, el uso de anotaciones de *strictness* reduce notablemente el consumo de heap y la duración de las fases de manejo de memoria de esta implementación. Como se verá en la Secc. 2.6, esto permite que el rendimiento obtenido sea comparable al de la implementación C.

2.4 Árboles AVL Imperativos

La implementación C de árboles AVL presenta grandes diferencias con respecto a las versiones funcionales: la estructura interna de los árboles está basada en *structs* y punteros, la implementación de las operaciones y del algoritmo de balance hace uso de efectos laterales y manipulación directa de punteros, y se utilizan punteros a función para emular polimorfismo paramétrico (característica no presentada por el lenguaje C). La idea es modelar los elementos del árbol mediante un tipo genérico (`void*`), y compararlos sólo mediante una función apropiada, provista por el usuario de la estructura de datos.

Otra diferencia a tener en cuenta es que esta versión, al contrario de las implementaciones funcionales, no está basada en pares clave-valor. Un nodo contiene únicamente el valor insertado en él. Las operaciones provistas son:

```
/* Crea un AVL vacio con una funcion de comparacion dada */
AVL* newAVL(int (*)(void*,void*));

/* Chequear si el AVL dado esta o no vacio */
int isEmptyAVL(AVL*);

/* Chequear si el elemento dado pertenece o no al AVL */
int isElemAVL(AVL*, void*);

/* Insertar un nuevo elemento en el AVL dado */
void insertAVL(AVL*, void*);

/* Borrar un elemento del AVL dado */
void deleteAVL(AVL*, void*);
```

Dado que esta tesis centra su estudio en las estructuras de datos funcionales, se omite el estudio en detalle de esta versión de árboles AVL. Sólo se dará la implementación del algoritmo de borrado y de una parte significativa del algoritmo de balance, con el objetivo de mostrar la complejidad añadida con respecto a las versiones funcionales. El código completo puede encontrarse en el CD adjunto a este trabajo.

2.4.1 Estructura Interna

La estructura interna está definida en dos niveles. El primero de ellos modela el árbol propiamente dicho; al igual que las versiones funcionales, un árbol AVL conoce su profundidad, el elemento almacenado en la raíz y sus dos subárboles, posiblemente vacíos (en este caso los punteros a dichos subárboles son igual a `NULL`). El segundo nivel es un *wrapper* sobre la definición del árbol destinada a proveer polimorfismo paramétrico, el cual se emula mediante una función para comparar a los elementos por mayor, menor o igual (la función deberá retornar un valor mayor, menor o igual a cero, respectivamente):

```
typedef struct AVLTree {
  int depth;
  void* elem;
  struct AVLTree *left, *right;
} AVLTree;

typedef struct AVL {
  int (*cmp)(void*,void*);
  struct AVLTree* avl;
} AVL;
```

2.4.2 Algoritmo de Borrado

Al igual que la implementación funcional de este algoritmo, la versión C divide el procedimiento en dos fases: buscar recursivamente el nodo v a eliminar , y borrarlo de acuerdo a los dos casos enumerados en la Secc. 2.2.3. En caso de que v no tenga subárboles vacíos, se invoca la función auxiliar `tspan`, la cual borra el predecesor inorder de v y lo retorna, de modo que pueda utilizarse para reemplazar a v en el árbol original. El balance del

árbol producido por `tspan` se asegura mediante una invocación a la función `balance`, la versión imperativa de *join* (Secc. 2.2.4).

```
void* tspan(AVLTree** avl)
{
  void* elem;

  if (isEmptyAVLTree((*avl)->right))
  {
    elem = (*avl)->elem;
    *avl = (*avl)->left;
    return elem;
  }
  else
  {
    elem = tspan(&((*avl)->right));
    balance(*avl);
    return elem;
  }
}
```

Dada esta función auxiliar, el algoritmo de borrado se implementa mediante la función `deleteAVL`, cuyo código se lista en la Fig. 2.3. Puede verse que la implementación procedural del algoritmo de borrado es mucho más compleja que la versión funcional pura (aunque sigue ocupando una página).

2.4.3 Algoritmo de Balance

La versión imperativa del algoritmo de balance se basa en el patrón implementado por la función *join* (dada en la Secc. 2.2.4), invocando en función de la diferencia de altura de los árboles a balancear a distintas funciones auxiliares que efectúan esta operación. No obstante, dado que se busca lograr un uso óptimo de la memoria, la implementación de la versión imperativa es mucho más complicada que la de las versiones funcionales. En efecto, la versión C hace uso *óptimo* del espacio de memoria, evitando crear nuevos nodos o estructuras intermedias. Esto se debe a la capacidad de los lenguajes procedurales de modificar *in-place* estructuras en memoria mediante

```
void deleteAVL(AVL* avl, void* elem)
{
  deleteAVLTree(&(avl->avl), elem, avl->cmp);
}

/* avl es una referencia a un puntero a un AVLTree */
void deleteAVLTree(AVLTree** avl, void* elem,
                   int (*cmp)(void*,void*))
{
  int res;

  if (isEmptyAVLTree(*avl))
    return;

  res = cmp(elem, (*avl)->elem);

  if (res < 0)
    deleteAVLTree(&((*avl)->left), elem, cmp);
  else if (res > 0)
    deleteAVLTree(&((*avl)->right), elem, cmp);
  else
    if (isEmptyAVLTree((*avl)->left)) {
      free(*avl);
      *avl = (*avl)->right;
    }
    else if (isEmptyAVLTree((*avl)->right)) {
      free(*avl);
      *avl = (*avl)->left;
    }
    else {
      void* elem = tspan(&((*avl)->left));
      (*avl)->elem = elem;
      balance(*avl);
    }
}
```

Figura 2.3: Función `deleteAVL`, versión imperativa

efectos laterales. El uso de efectos laterales complica también la demostración de la correctitud del algoritmo, la cual ya no puede efectuarse mediante razonamiento ecuacional.

El algoritmo de balance está implementado por la función `balance`. Según la diferencia de profundidad de los árboles a balancear, invoca a diferentes funciones auxiliares que realizan el balance:

```
void balance(AVLTree* avl)
{
  int ld = depth(avl->left);
  int rd = depth(avl->right);

  if (abs(ld - rd) <= 1)
    setDepth(avl);
  else if (ld == rd + 2)
    leftBalance(avl);
  else if (ld + 2 == rd)
    rightBalance(avl);
  else if (ld > rd + 2)
    recBalanceL(avl);
  else if (ld + 2 < rd)
    recBalanceR(avl);
}
```

Las funciones `recBalanceL` y `recBalanceR` representan los casos recursivos de la función *join*. Pero a diferencia de dicha función, los casos recursivos de la versión procedural no crean en memoria nuevos nodos ni estructuras intermedias. Lo mismo sucede con las funciones `leftBalance` y `rightBalance`, equivalentes a *ljoin* y *rjoin*, respectivamente. Esto puede apreciarse a partir de la Fig. 2.4, donde se lista el código de la función `leftBalance` (el código de `rightBalance` es dual).

Las funciones `leftBalanceL` y `leftBalanceR` corresponden a las dos alternativas de la función *ljoin* según sea el hijo izquierdo del subárbol izquierdo más profundo o no que el hijo derecho de ese subárbol, respectivamente. Puede apreciarse que la reorganización del árbol se efectúa mediante la manipulación de los distintos componentes de su estructura interna, evitando la construcción de nuevos nodos o la duplicación de los ya existentes. Este

```
void leftBalance(AVLTree* avl) {
  if (depth(avl->left->left) >= depth(avl->left->right))
    leftBalanceL(avl);
  else
    leftBalanceR(avl);
}

void leftBalanceL(AVLTree* avl) {
  AVLTree* l = avl->left;
  void* aux  = avl->elem;

  avl->elem  = l->elem;
  l->elem    = aux;
  avl->left  = l->left;
  l->left    = l->right;
  l->right   = avl->right;
  avl->right = l;
  setDepth(avl->right);
  setDepth(avl);
}

void leftBalanceR(AVLTree* avl) {
  AVLTree* r  = avl->right;
  AVLTree* lr = avl->left->right;
  void *aux   = avl->elem;

  avl->elem = lr->elem;
  lr->elem  = aux;
  avl->left->right = lr->left;
  avl->right = lr;
  avl->right->left  = lr->right;
  avl->right->right = r;
  setDepth(avl->left);
  setDepth(avl->right);
  setDepth(avl);
}
```

Figura 2.4: Función `leftBalance`

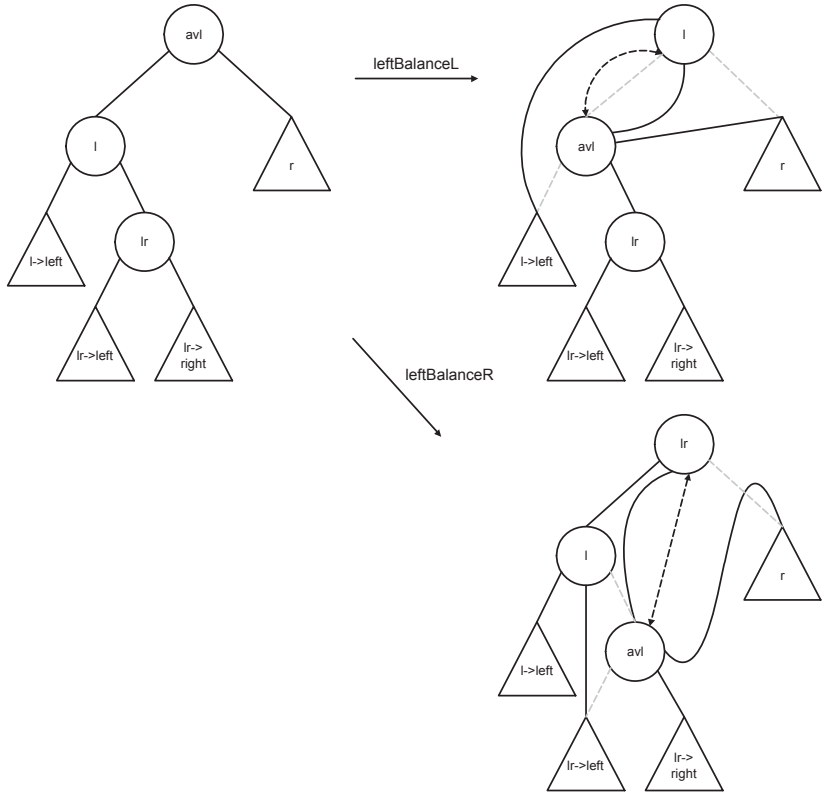

Figura 2.5: Reorganización producida por `leftBalance`

proceso se ilustra en la Fig. 2.5. Las líneas punteadas negras indican un intercambio entre los contenidos de dos nodos, y las grises muestran la vieja estructura de árbol. Obsérvese que los árboles obtenidos mediante estas dos funciones tienen la misma organización que los resultantes en la Fig. 2.2.

2.5 Correctitud

Se demostrará que las funciones *insertAVL* y *deleteAVL* preservan tanto la condición de árbol de búsqueda como la condición de balance. La prueba se vale de la siguiente notación introducida en la Secc. 2.1, donde para todo árbol t, se define a d_t como la profundidad de t.

A partir de las figuras 2.1 y 2.2 (Secc. 2.2.4) puede observarse que las funciones que implementan el algoritmo de balance reorganizan los subárboles de modo tal que todo subárbol mantiene la posición (a la derecha o a la izquierda) con respecto al padre que tenía antes de la reorganización, por lo

que se preserva la condición de árbol de búsqueda. De la misma manera, es sencillo verificar que *insertAVL* y *deleteAVL* mantienen dicha condición.

Mediante inducción sobre la estructura del árbol puede demostrarse que las funciones *insertAVL* y *deleteAVL* mantienen la condición de balance siempre que la función *join* produzca árboles balanceados, donde un árbol está balanceado sii satisface el predicado *isBalanced*. Luego, la correctitud de las funciones *insertAVL* y *deleteAVL* reduce a la correctitud de la función *join*. Antes de efectuar la demostración, se probará el siguiente resultado auxiliar:

Lema 2.5.1 *Sean l y r dos árboles balanceados tq. $|\, d_l - d_r \,| \leqslant 1$. Sea t = sjoin l k v r. Entonces, t es un árbol balanceado con $d_t = 1 + max\{d_l, d_r\}$, para todo par clave-valor (k,v).*

Demostración: A partir código de la función *sjoin* (Secc. 2.2.4), como l y r están balanceados y $|\, d_l - d_r \,| \leqslant 1$, t satisface *isBalanced*. Además, como *sjoin* agrega un nivel de profundidad, $d_t = 1 + max\{d_l, d_r\}$. Queda entonces demostrado el lema 2.5.1 $\square$

Teorema 2.5.2 *Sean l y r árboles balanceados. Sea t = join l k v r . Entonces, t es un árbol balanceado con $max\{d_l, d_r\} \leqslant d_t \leqslant 1 + max\{d_l, d_r\}$, para todo par clave-valor (k,v).*

Demostración: Se hará por inducción sobre el valor absoluto de la diferencia de altura entre l y r. Para el caso base se presentan tres alternativas posibles:

1. $|\, d_l - d_r \,| \leqslant 1$:
   En este caso, t = *sjoin l k v r*. Por el lema 2.5.1, t satisface el invariante *isBalanced* y $d_t = 1 + max\{d_l, d_r\}$.

2. $d_l = d_r + 2$:
   En este caso, t = *ljoin l k v r*. Sea l = *Node d_l ll lk lv lr*. A partir del código de *ljoin*, existen dos casos para analizar (ver la Fig. 2.2):

   (a) $d_{ll} \geqslant d_{lr}$:
   Entonces, *ljoin l k v r* = *sjoin ll lk lv* (*sjoin lr k v r*). Como por hipótesis l está balanceado,

$$
\begin{array}{ll}
 & d_{ll} - 1 \leqslant d_{lr} \leqslant d_{ll} \\
\equiv & \quad \{d_{ll} = d_l - 1, \text{ por ser } d_{ll} \text{ el hijo más profundo de } l\} \\
 & d_l - 2 \leqslant d_{lr} \leqslant d_l - 1 \\
\equiv & \quad \{d_l = d_r + 2, \text{ por hipótesis}\} \\
 & d_r \leqslant d_{lr} \leqslant d_r + 1
\end{array}
$$

Por el lema 2.5.1, $t_1 = sjoin\ lr\ k\ v\ r$ es un árbol balanceado con $d_l - 1 = d_r + 1 \leqslant d_{t_1} \leqslant d_r + 2 = d_l$. Como $d_{ll} = d_l - 1$, otra vez por el lema 2.5.1 se tiene que $t = sjoin\ ll\ lk\ lv\ t_1$ es un árbol balanceado con $d_l \leqslant d_t \leqslant d_l + 1$. Como $max\{d_l, d_r\} = d_l$, queda probada esta alternativa.

(b) $d_{ll} < d_{lr}$:
Como $d_l \geqslant 2$, lr no está vacío. Sea $lr = Node\ d_{lr}\ lrl\ lrk\ lrv\ lrr$. Luego, $ljoin\ l\ k\ v\ r = sjoin\ (sjoin\ ll\ lk\ lv\ lrl)\ lrk\ lrv\ (sjoin\ lrr\ k\ v\ r)$. Como lr está balanceado,

$$\begin{array}{ll} & d_{lr} - 2 \leqslant d_{lrl} \leqslant d_{lr} - 1 \\ \equiv & \quad \{d_{lr} = d_{ll} + 1, \text{ por ser } d_{lr} \text{ el hijo más profundo de } l\} \\ & d_{ll} - 1 \leqslant d_{lrl} \leqslant d_{ll} \end{array}$$

Por el lema 2.5.1, $t_1 = sjoin\ ll\ lk\ lv\ lrl$ está balanceado, y $d_{t_1} = d_{ll} + 1$. Del mismo modo,

$$\begin{array}{ll} & d_{lr} - 2 \leqslant d_{lrr} \leqslant d_{lr} - 1 \\ \equiv & \quad \{d_{lr} = d_l - 1, \text{ por ser } d_{lr} \text{ es el hijo más profundo de } l\} \\ & d_l - 3 \leqslant d_{lrr} \leqslant d_l - 2 \\ \equiv & \quad \{d_l = d_r + 2, \text{ por hipótesis}\} \\ & d_r - 1 \leqslant d_{lrr} \leqslant d_r \end{array}$$

Por el lema 2.5.1, $t_2 = sjoin\ lrr\ k\ v\ r$ está balanceado, y $d_{t_2} = d_r + 1$. Como $d_{t_1} = d_{ll} + 1 = d_{lr} = d_l - 1 = d_r + 1 = d_{t_2}$, por el lema 2.5.1, $t = sjoin\ t_1\ lrk\ lrv\ t_2$ es un árbol balanceado con $d_t = d_r + 2 = d_l$. Dado que $d_l = max\{d_l, d_r\}$, queda probada esta alternativa.

3. $d_l + 2 = d_r$:
La demostración de este caso es dual al anterior, por lo tanto se omite de la demostración general.

HI: Sean l y r árboles balanceados con $|\, d_l - d_r \,| < h$, con $h \leqslant 1$. Entonces, $join\ l\ k\ v\ r$ satisface *isBalanced*, para todo par clave–valor (k, v).

TI: Sean l y r árboles balanceados con $|\, d_l - d_r \,| = h$, con $h > 2$. Entonces, $join\ l\ k\ v\ r$ satisface *isBalanced*, para todo par clave–valor (k, v). Para probar la tesis inductiva deben analizarse las siguientes alternativas (ver la Fig. 2.1):

1. $d_l > d_r + 2$:
Sea $l = Node\ d_l\ lr\ lk\ lv\ lr$. En este caso, $join\ l\ k\ v\ r = join\ ll\ lk\ lv\ (join\ lr$

$k\ v\ r$). Como $d_l - 2 \leqslant d_{lr} \leqslant d_l - 1$, $h-2 \leqslant |\, d_{lr} - d_r \,| \leqslant h-1$. Luego, por HI, $t_1 = join\ lr\ k\ v\ r$ es un árbol balanceado con $d_{lr} \leqslant d_{t_1} \leqslant d_{lr} + 1$, ya que $d_{lr} > d_r$. Dado que l está balanceado, $|\, d_{ll} - d_{lr} \,| \leqslant 1$, luego $0 \leqslant |\, d_{ll} - d_{t_1} \,| \leqslant 2$ y por HI $t = join\ ll\ lk\ lv\ t_1$ está balanceado. Queda por analizar la profundidad de t. Hay dos casos posibles:

(a) $d_{t_1} = d_{lr}$:
Dado que l está balanceado, $t = join\ ll\ lk\ lv\ t_1 = sjoin\ ll\ lk\ lv\ t_1$ y $d_t = 1 + max\{d_{ll}, d_{lr}\} = d_l$.

(b) $d_{t_1} = d_{lr} + 1$:
Si $d_{ll} \geqslant d_{lr}$, otra vez *join* es igual a *sjoin* y $d_t = 1 + max\{d_{ll}, d_{lr}\} = d_l$. Si $d_{ll} < d_{lr}$, $d_{ll} + 2 = d_{t_1}$ y *join* es igual a *rjoin*. Entonces, $d_l = d_{lr} + 1 \leqslant d_t \leqslant d_{lr} + 2 = d_l + 1$.

2. $d_l + 2 < d_r$:
La demostración de este caso es dual al anterior, por lo tanto se omite de la demostración general.

Queda así completada la demostración de teorema 2.5.2 □

El teorema 2.5.2 es el componente principal de la demostración de la correctitud de las implementaciones Haskell y Clean de árboles AVL. Para obtener una prueba similar correspondiente a la versión procedural hay que razonar sobre una lógica de programación basada por ejemplo, en el sistema formal $\mathcal{H}$ [Hoa69], lo que hace la demostración mucho más extensa y complicada. Este hecho es generalizable: los lenguajes imperativos no permiten que sus expresiones se traten de manera matemática debido a que, al contrario que los lenguajes funcionales puros, presentan efectos laterales. En consecuencia, se debe razonar sobre los programas imperativos de una manera "artificial", lo que dificulta las demostraciones de correctitud parcial y total. Este es uno de los principales argumentos que compensan la eficiencia levemente inferior de los lenguajes funcionales con respecto a los imperativos.

2.6 Análisis de Eficiencia

La eficiencia de las implementaciones de árboles AVL discutidas a lo largo de este capítulo se probó mediante la inserción de 10^3, 10^4 y 10^5 elementos generados pseudo–aleatoriamente, seguidas por una función que computa la altura del árbol resultante. Esta última función se incluye para forzar a las implementaciones Haskell y Clean, que son *lazy*, a computar las inserciones

Versión	10^3	10^4	10^5
Haskell	934.80 Kb	10392.69 Kb	114742.69 Kb
Clean	358.56 Kb	4519.36 Kb	54612.78 Kb
Ratio (%)	38.35%	43.48%	47.49%
Mejora	61.65%	56.52%	52.51%

Tabla 2.1: Consumo de heap de la función *insertAVL*

necesarias para llevar a cabo la prueba. Como el cálculo de la altura de un árbol requiere recorrer todos sus nodos, se garantiza así la inserción de la totalidad de los elementos.

Las pruebas fueron efectuadas bajo el sistema operativo Linux, utilizando un heap de 60 Mbytes. Los tiempos de ejecución y datos sobre el consumo de memoria para la versión Haskell fueron obtenidos mediante las herramientas de *profiling* provistas por dicho compilador. Dado que Clean no provee *profilers* para Linux, los tiempos se obtuvieron mediante los reportes producidos por el programa una vez finalizada su ejecución, mientras que el consumo de memoria fue analizado bajo Win32. No obstante, la experiencia muestra que el rendimiento bajo ambas plataformas es similar, por lo que los resultados obtenidos son aplicables a Linux.

2.6.1 Eficiencia Espacial

Las pruebas efectuadas muestran que hay diferencias de performance importantes entre ambas versiones funcionales. El consumo de memoria por parte de la implementación Haskell es mucho mayor al presentado por la versión Clean. Esta diferencia obedece fundamentalmente al uso de las anotaciones de *strictness* provistas por Clean.

La Tabla 2.1 lista el rendimiento espacial de las versiones Clean y Haskell. El uso de memoria por parte de la implementación Clean representa entre el 38% y el 47% del heap utilizado por la versión Haskell, lo que implica una mejora que oscila entre el 53% y el 62% según el tamaño de entrada considerada. La Fig. 2.6 muestra un gráfico comparativo para los valores listados en la Tabla 2.1. Debido a que el eje x crece según una progresión aritmética de factor 10, se utilizan coordenadas logarítmicas donde el eje y crece en la misma proporción, de modo que el gráfico refleje la verdadera forma de la curvas obtenidas.

Un árbol con n elementos tiene esa misma cantidad de nodos. Luego, la complejidad espacial de los árboles AVL es $O(n)$. Para analizar la relación entre los consumos de memoria obtenidos y esta fórmula, se analizará los

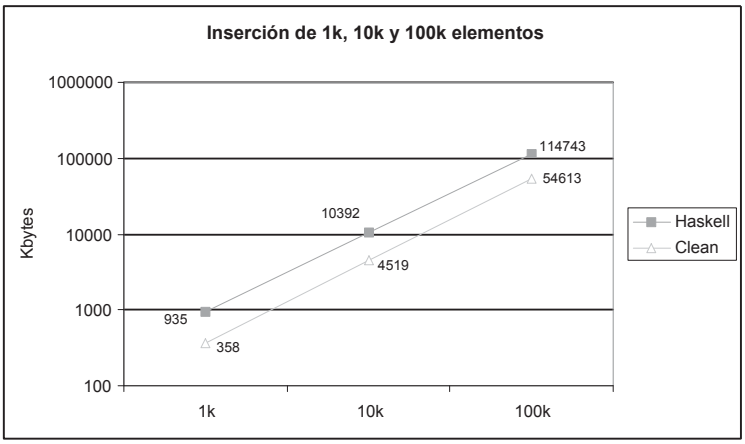

Figura 2.6: Gráfico del uso de heap dado en la Tabla 2.1

$f_S(i)$	$i = 3$	$i = 4$	$i = 5$
Haskell	0.934	1.039	1.147
Clean	0.358	0.451	0.546

Tabla 2.2: Constantes de proporcionalidad de complejidad espacial

resultados de la función f_S, definida como $f_S = \frac{S(i)}{10^i}$, $i \in \{3, 4, 5\}$, donde $S(i)$ se define como el consumo de memoria resultante de insertar 10^i elementos. Como f_S computa para cada versión una aproximación del valor de la constante de proporcionalidad, debería aproximarse a una función constante para cada implementación. La Tabla 2.2 muestra los valores tomados por f_S. Si bien están próximos entre sí, f_S presenta una tendencia levemente creciente. Esto puede deberse a que algunos nodos son aún referenciados al momento de ejecutarse las rutinas de manejo automático de memoria (*garbage collection*), por lo que permanecen en el heap. Notar también que los valores de la constante de proporcionalidad de la versión Clean son menores que la mitad de los obtenidos para la implementación Haskell.

2.6.2 Eficiencia Temporal

El rendimiento temporal de las versiones funcionales se ve influenciado por el espacial, ya que un programa con un consumo de memoria alto debe suspender su ejecución para invocar a una fase de *garbage collection*, de modo que se eliminen del heap los elementos que ya no son referenciados. De este modo, el tiempo de ejecución de la versión Haskell es mayor al de la versión Clean, ya que Haskell pasa más tiempo administrando memoria.

Versión	10^3	10^4	10^5
Haskell	20 ms	260 ms	3760 ms
Clean	7 ms	80 ms	1150 ms
C	6 ms	80 ms	1050 ms

Tabla 2.3: Tiempos de ejecución de la función *insertAVL*

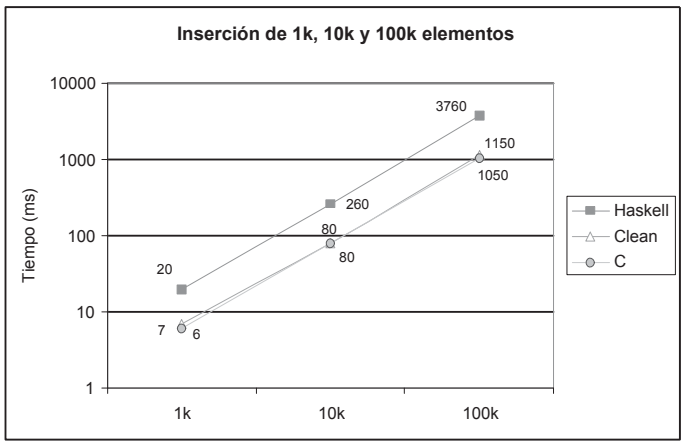

Figura 2.7: Gráfico de los tiempos de ejecución de la Tabla 2.3

Los resultados obtenidos se listan en la Tabla 2.3. Con respecto a los tiempos de ejecución obtenidos para la versión C, la implementación Haskell tarda sólo 3.33, 3.25 y 3.58 veces más para cada una de las entradas, en orden creciente. La implementación Clean es aún más eficiente. Para 10^3 elementos tarda 1.16 veces más que la versión C, y 1.09 veces para 10^5 elementos. El mejor resultado fue el obtenido para la inserción de 10^4 elementos: en este caso, el tiempo de ejecución iguala al obtenido por el lenguaje C.

La Fig. 2.7 muestra un gráfico comparativo de los tiempos de ejecución. Al igual que con la Fig. 2.6, se utilizan coordenadas logarítmicas para reflejar la verdadera forma del gráfico. Notar la cercanía de la curva correspondiente a la versión Clean con respecto a la obtenida por la implementación C.

Para comparar los tiempos resultantes se utilizará el mismo mecanismo que en la Secc. 2.6.1. O sea, se analizará el comportamiento de la función $f_T = 10^3 \frac{T(i)}{10^i \log(10^i)}$, $i \in \{3, 4, 5\}$, donde $T(i)$ se define como el tiempo obtenido para la inserción de 10^i elementos. El producto por 10^3 es para normalizar los valores retornados por f_T, evitando que sean muy reducidos y que se pierda precisión por el sistema de numeración de punto flotante. Dado que la complejidad de realizar n inserciones en un árbol AVL es $O(n \log(n))$, f_T debería aproximarse a una función constante para cada implementación. La

$f_T(i)$	$i = 3$	$i = 4$	$i = 5$
Haskell	2.006	1.956	2.263
Clean	0.702	0.602	0.692
C	0.602	0.602	0.632

Tabla 2.4: Constantes de proporcionalidad de complejidad temporal

Tabla 2.4 lista los valores que toma esta función. Puede verse que en general f_T es una función constante, de acuerdo a lo esperado.

2.6.3 Conclusiones

Los valores obtenidos para las pruebas de rendimiento espacial y temporal (en especial las constantes de proporcionalidad) permiten concluir que Clean posee un grado de eficiencia considerablemente superior al presentado por Haskell. Esta diferencia puede atribuirse a las *strictness annotations* y a la implementación sumamente eficiente que presenta dicho lenguaje.

Los tiempos de ejecución dados en la sección anterior muestran que es posible implementar eficientemente estructuras de datos funcionales puras, obteniendo rendimientos comparables a los alcanzados por un lenguaje imperativo. Esto vale incluso para entradas de datos grandes, del orden de los 10^5 elementos. Los factores predominantes para lograr eficiencia son:

- El uso de técnicas de *profiling*, mediante las cuales es posible identificar con precisión los fragmentos de código que provocan un rendimiento ineficiente.

- El uso de mecanismos tales como las anotaciones de *strictness* para limitar el uso de la *lazy evaluation* al dominio de la aplicación donde es realmente necesaria, lo que permite controlar de manera indirecta el consumo de memoria. Una ventaja adicional de este mecanismo es su carácter "no intrusivo": en general, las anotaciones de *strictness* no disminuyen la claridad los programas que las utilizan.

Otro detalle que debe ser considerado es que C no implementa *garbage collection*. A diferencia de las implementaciones funcionales, la versión imperativa no posee manejo automático de memoria, o sea, la única manera de liberar memoria es explícitamente. Como los casos de pruebas efectuados sólo insertan nodos, el análisis de eficiencia tiende a favorecer a la implementación C, ya que no consume tiempo liberando la memoria utilizada. Esto hace aún más valiosos los resultados obtenidos por las versiones Haskell y Clean.

Capítulo 3

Treaps

Las técnicas para mantener a un árbol de búsqueda balanceado pueden ser complicadas, particularmente si se implementan mediante algoritmos imperativos. En algunas ocasiones resulta conveniente sacrificar un balance óptimo en favor de una implementación sencilla que asegure uno "razonable", o sea, cercano al óptimo. Esto es, para un árbol binario de búsqueda t con n nodos, se desea que d_t sea en promedio $O(\log(n))$ en lugar de *exactamente* $\log(n)$. Al igual que en el capítulo anterior, d_t representa la altura de un árbol t.

Los treaps [AS89] ofrecen una solución simple al problema de mantener el balance. La idea es simular que los elementos que conforman un treap fueron insertados de manera aleatoria. Dado que la profundidad esperada de un árbol t cuyos elementos fueron insertados al azar es $O(\log(n))$ [AHU83, SF95], esta técnica permite obtener el efecto buscado.

Al igual que con los árboles AVL, se mostrará que los lenguajes funcionales puros permiten implementar treaps de manera sumamente elegante, y con eficiencia similar a la alcanzable por un lenguaje imperativo.

La organización de este capítulo es la siguiente. La Secc. 3.1 da la definición de treap. Las Secc. 3.2 y 3.3 estudian las implementaciones Haskell y Clean de treaps, respectivamente. La versión C se discute en la Secc. 3.4. En la Secc. 3.5 se prueba formalmente la correctitud de las dos versiones funcionales. Finalmente, la Secc. 3.6, estudia y compara la eficiencia de las tres implementaciones presentadas, mostrando que la eficiencia de las versiones funcionales es comparable a la obtenida por la versión C.

3.1 Definición de Treap

Un treap es un árbol de búsqueda donde el balance se asegura simulando que los elementos que lo componen fueron insertados en orden aleatorio. Es-

te efecto de aleatoridad se logra asociando una *prioridad* generada al azar a todo nodo del árbol, y agregando una condición extra que deberá cumplirse simultáneamente con la de árbol búsqueda: la prioridad de un nodo debe ser menor que la de la raíz de sus subárboles, si existen. Notar que esta condición coincide exactamente con la definición de *min-heap* [AHU83]. Para formalizarla, se definirá un predicado llamado *minHeap*, que deberá cumplirse para todo treap t bien formado. Una primera aproximación a esta definición consiste en separarla en dos casos posibles:

- Si t es una hoja, satisface *minHeap*
- En caso contrario, t satisface *minHeap* si sus subárboles lo satisfacen y la prioridad de su nodo raíz es menor o igual que la de sus hijos.

Para dar un predicado que represente a la formalización de estas dos alternativas se utilizará la notación introducida en la Secc. 2.1. Para todo árbol t, $t.l$ representa a su subárbol izquierdo, y $t.r$ al derecho. Además, t_p simboliza la prioridad del nodo raíz de t. Se asumirá que si t está vacío, $t_p = +\infty$.

$$\begin{array}{l} \mathit{minHeap}(t) \equiv \\ \quad d_t = 0 \vee (\mathit{minHeap}(t.l) \wedge \mathit{minHeap}(t.r) \wedge t_p \leqslant \mathit{min}\{t.l_p, t.r_p\}) \end{array}$$

Una implementación correcta de treaps debe mantener la invariancia de este predicado para todo treap bien formado. La formulación formal de *minHeap* será de utilidad en la Secc. 3.5 para probar la correctitud de las implementaciones funcionales. La Fig. 3.1 de un ejemplo de un treap. El valor almacenado por cada nodo está en itálica y las prioridades asociadas se muestran entre paréntesis. Notar que si bien se satisfacen la condición de búsqueda y el predicado *minHeap*, esto no necesariamente implica que el árbol está balanceado.

3.2 Treaps: Versión Haskell

La versión Haskell de treaps se compone de dos partes: la implementación de las operaciones básicas sobre esta estructura de datos, tales como búsqueda, inserción y borrado, y las funciones auxiliares que aseguran *minHeap*. Las operaciones básicas se basan en estas últimas, denominadas *operaciones de rotación*. Para el estudio de la versión Haskell se discutirán las funciones básicas y luego se analizarán las operaciones de rotación; previamente, se dará la signatura de las operaciones principales sobre un treap y la estructura interna utilizada para su implementación.

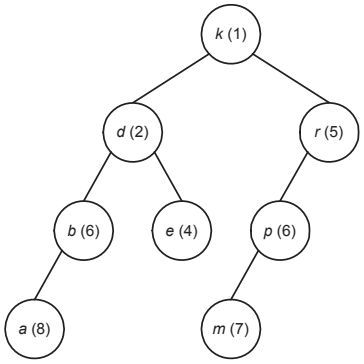

Figura 3.1: Un treap

3.2.1 Operaciones

La implementación de treaps provee la funcionalidad típica de los árboles de búsqueda. Las operaciones principales son:

$$
\begin{array}{lll}
newTreap & :: & Ord\ a \Rightarrow Integer \rightarrow Treap\ a \\
isElemT & :: & Ord\ a \Rightarrow Treap\ a \rightarrow a \rightarrow Bool \\
insertT & :: & Ord\ a \Rightarrow Treap\ a \rightarrow a \rightarrow Treap\ a \\
deleteT & :: & Ord\ a \Rightarrow Treap\ a \rightarrow a \rightarrow Treap\ a
\end{array}
$$

El contexto *Ord a* expresa que los elementos del tipo *a* deben ser comparables según un relación de orden. Este requerimiento es necesario porque todo treap es también un árbol de búsqueda, y debe existir una relación de orden total sobre sus claves.

3.2.2 Estructura Interna

Un treap asocia un valor de prioridad aleatorio a cada elemento almacenado. Para modelar esta asociación se utilizará un registro. En cuanto al árbol, una manera conveniente de utilizar números aleatorios en Haskell es mediante una lista infinita de tales números, la cual se puede producir a partir de una función perteneciente a la biblioteca standard del lenguaje. Por lo tanto, un treap puede definirse como un árbol binario más un suministro infinito de número aleatorios:

---

data $TVal\ a\ =\ TVal\ \{\ priority :: Int, value :: a\ \}$

data $Ord\ a\ \Rightarrow\ BTree\ a\ =\ EmptyT\ \mid\ Node\ a\ (BTree\ a)\ (BTree\ a)$
newtype $Ord\ a\ \Rightarrow\ Treap\ a\ =\ Treap\ ([Int], BTree\ (TVal\ a))$

Puede verse la naturaleza inherentemente lazy de esta estructura de datos. Si la lista infinita fuese estricta, su computación jamás finalizaría. Por este mismo motivo, es necesario tener la precaución de utilizar la lista siempre en un contexto *lazy* (p. ej., no se puede obtener la longitud de la lista, pero sí cualquier segmento inicial finito). Notar también que para las comparaciones entre elementos de tipo $(TVal\ a)$ se debe usar el campo *value*.

3.2.3 Búsqueda, Inserción, Borrado

El algoritmo para verificar si un valor dado pertenece o no a un treap es idéntico al de un árbol de búsqueda ordinario, por lo que será omitido.

Para insertar un elemento en un treap vacío, se debe crear una hoja donde se guardará el elemento junto a su prioridad asociada. En caso contrario, se inserta recursivamente sobre el subárbol izquierdo o derecho, preservando la condición de árbol de búsqueda. Para mantener *minHeap* es necesario verificar que la prioridad del nodo insertado sea mayor que la de su nodo padre. Esta tarea la realiza la función *checkLeft* si la inserción se efectuó en el subárbol izquierdo, o *checkRight* en el otro caso. Si el invariante no se cumple, invocan a las operaciones de rotación, *rotateRight* o *rotateLeft* respectivamente. Observar que se asocia el elemento a insertar con su prioridad, la cual se obtiene de la cabeza de la lista infinita que suministra los números aleatorios, mientras que la cola permanece sin evaluar.

$insertT\ ::\ Ord\ a\ \Rightarrow\ Treap\ a\ \rightarrow\ a\ \rightarrow\ Treap\ a$
$insertT\ (Treap\ (p{:}ps,\ t))\ v\ =\ Treap\ (ps,\ insertBT\ t\ (TVal\ p\ v))$

$insertBT\ ::\ Ord\ a\ \Rightarrow\ BTree\ (TVal\ a)\ \rightarrow\ TVal\ a\ \rightarrow\ BTree\ (TVal\ a)$
$insertBT\ EmptyT\ v_i\ =\ Node\ v_i\ EmptyT\ EmptyT$
$insertBT\ (Node\ v\ tl\ tr)\ v_i$
$\mid\ v_i\ <\ v\ \ \ \ \ = checkLeft\ \ (Node\ v\ (insertBT\ tl\ v_i)\ tr)$
$\mid$ **otherwise** $= checkRight\ (Node\ v\ tl\ (insertBT\ tr\ v_i))$

```
checkLeft :: BTree (TVal a) → BTree (TVal a)
checkLeft t@(Node v (Node v_l _ _ ) _ )
 | priority v > priority v_l = rotateRight t
 | otherwise                 = t

checkRight :: BTree (TVal a) → BTree (TVal a)
checkRight t@(Node v _ (Node v_r _ _ ))
 | priority v > priority v_r = rotateLeft t
 | otherwise                 = t
```

El borrado es la operación inversa de la inserción no sólo a nivel semántico, sino también operacionalmente. O sea, si alguno de los subárboles del nodo a eliminar es vacío, se debe retornar el otro. En caso contrario se debe efectuar una rotación de modo que dicho nodo descienda un nivel. De este modo, el algoritmo puede aplicarse recursivamente. Notar que el nodo descendido un nivel tendrá prioridad más baja que su padre, por lo que se rompe el invariante *minHeap*. Sin embargo, una vez que se alcance el caso base dicho nodo es eliminado del treap, y el invariante vuelve a satisfacerse. El algoritmo resultante es el siguiente:

```
deleteT :: Ord a ⇒ Treap a → a → Treap a
deleteT (Treap (ps, t)) v = Treap (ps, deleteBT t v)

deleteBT :: Ord a ⇒ BTree (TVal a) → a → BTree (TVal a)
deleteBT EmptyT _ = EmptyT
deleteBT t@(Node v tl tr) v_d
 | v_d < value v = Node v (deleteBT tl v_d) tr
 | v_d > value v = Node v tl (deleteBT tr v_d)
 | otherwise     = deleteBT' t

deleteBT' :: BTree (TVal a) → BTree (TVal a)
deleteBT' (Node _ tl EmptyT) = tl
deleteBT' (Node _ EmptyT tr) = tr
deleteBT' t@(Node _ (Node v_l _ _ )(Node v_r _ _ ))
 | priority v_l ⩽ priority v_r = let (Node v_n nl nr) = rotateRight t
                                 in Node v_n nl (deleteBT' nr)
 | otherwise                   = let (Node v_n nl nr) = rotateLeft t
                                 in Node v_n (deleteBT' nl) nr
```

3.2.4 Operaciones de Rotación

Las operaciones de rotación son utilizadas con fines diferentes según sean invocadas por las funciones *insertT* o *deleteT*: para una inserción aseguran *minHeap*; para una operación de borrado permiten descender un nivel el nodo a borrar en caso que ninguno de sus subárboles esté vacío.

Luego del paso recursivo de una inserción sobre un treap t, puede suceder que la raíz v_i del subárbol t_i donde se realizó la operación (que, inductivamente, será un treap) tenga una prioridad menor que la del nodo padre. En este caso, se debe rotar a t de modo que la raíz del árbol resultante pase a ser v_i, la raíz de t sea uno de los hijos de v_i, y que la condición de búsqueda sea mantenida. Si t_i es el subárbol izquierdo, se debe efectuar una rotación a derecha (*rotateRight*); en caso contrario, se debe efectuar una rotación a izquierda (*rotateLeft*). La Fig. 3.2 ilustra la reorganización producida por estas dos operaciones. Puede verse que ambas funciones preservan la condición de búsqueda, dado que después de ser aplicadas todo subárbol continúa estando a la derecha o a la izquierda de su nodo padre original.

$$
\begin{aligned}
&rotateRight :: BTree\ a \rightarrow BTree\ a \\
&rotateRight\ (Node\ v\ (Node\ v_l\ ll\ lr)\ r) = Node\ v_l\ ll\ (Node\ v\ lr\ r)
\end{aligned}
$$

$$
\begin{aligned}
&rotateLeft :: BTree\ a \rightarrow BTree\ a \\
&rotateLeft\ (Node\ v\ l\ (Node\ v_r\ rl\ rr)) = Node\ v_r\ (Node\ v\ l\ rl)\ rr
\end{aligned}
$$

Para eliminar un nodo interior es necesario transformarlo en una hoja. Esta tarea puede lograrse mediante las operaciones de rotación, ya que ambas descienden un nivel la raíz del árbol rotado. De este modo, si la prioridad de la raíz del subárbol izquierdo del nodo a borrar es menor o igual que la del derecho se debe efectuar una rotación a derecha, o a izquierda en caso contrario. Si bien el árbol resultante no es un treap ya que el nodo descendido tiene prioridad menor que el nuevo padre, dicho nodo será eliminado y el resultado final sí será un treap válido.

Al igual que las funciones de balance para árboles AVL (Secc. 2.2.5), las operaciones de rotación son espacialmente ineficientes, ya que no pueden reorganizar un árbol sin crear un nuevos nodos. El motivo es el mismo, esto es, no puede actualizarse *in-place* una estructura sin correr el riesgo de introducir efectos laterales. El ejemplo dado en aquella sección es aplicable también en este caso, sólo basta reemplazar *join* por *rotateLeft* o *rotateRight*. Se verá

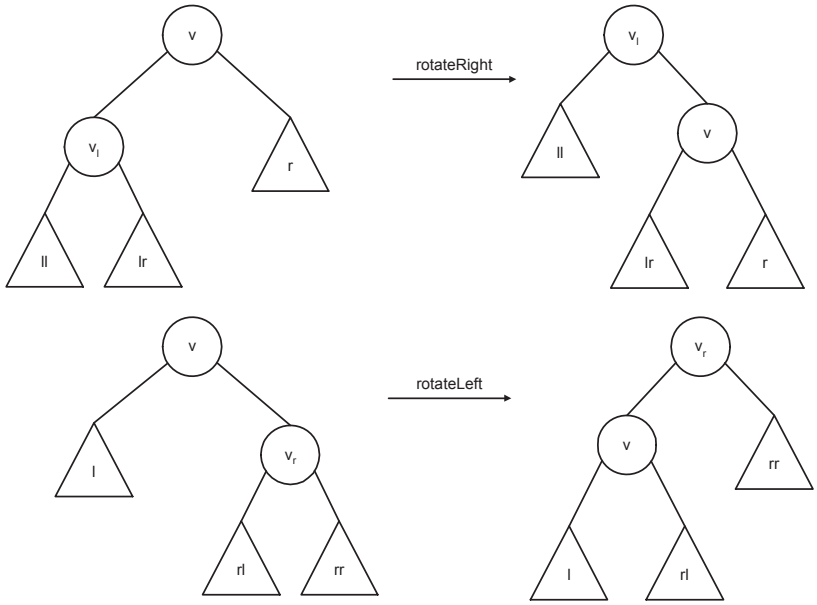

Figura 3.2: Operaciones de rotación

que este problema no ocurre con la versión C de dichas operaciones, dada en la Secc. 3.4. Dicha versión simplemente reorganiza punteros, reutilizando así los nodos ya existentes.

3.3 Treaps: Versión Clean

De la misma manera que la implementación Clean de árboles AVL (Secc. 2.3), la versión Clean de treaps hace uso de la capacidad del lenguaje para declarar anotaciones de *strictness* de manera explícita, lo que evita la proliferación de *closures* y provoca una disminución en el consumo de heap. Utilizando anotaciones de *strictness*, el tipo algebraico que define a los treaps es el siguiente:

$$:: \; Treap\ a \;=\; Treap\ ([\,!Int],\ !BTree\ (TVal\ a))$$

$$:: \; BTree\ a \;=\; EmptyT \mid Node\ a\ !(BTree\ a)\ !(BTree\ a)$$

$$:: \; TVal\ a \;=\; \{\ priority\ :: !Int,\ value\ ::\ a\ \}$$

Un treap es un árbol binario estricto de registros del tipo ($TVal\ a$), más un suministro infinito de números aleatorios. Si bien la lista debe ser *lazy* porque es infinita, sus elementos son estrictos pues es más eficiente mantener en el heap una lista de números enteros que una de *closures*. El tipo ($TVal\ a$) modela a un registro compuesto por dos campos: la prioridad aleatoria, declarada estricta, y el valor asociado, que permanece *lazy* de modo que sea posible almacenar expresiones que requieren de la *lazy evaluation* para que un programa no falle (por ejemplo, estructuras infinitas). El árbol binario declara estrictos a sus dos subárboles, de esta manera este atributo se propaga recursivamente hasta las hojas.

Las anotaciones de *strictness* también fueron utilizadas en la signatura de tipo de las operaciones sobre treaps, que en Clean quedan definidas así:

$$
\begin{array}{lll}
newTreap & :: !Int \rightarrow Treap\ a & \mid < a \\
isElemT & :: !(Treap\ a)\ a \rightarrow Bool & \mid < a \\
insertT & :: !(Treap\ a)\ a \rightarrow Treap\ a & \mid < a \\
deleteT & :: !(Treap\ a)\ a \rightarrow Treap\ a & \mid < a
\end{array}
$$

Como sucede con la implementación Haskell, el tipo a debe ser comparable, restricción que en Clean se expresa la notación ($\mid < a$). Para todas las operaciones, se declara estricto el treap pasado como parámetro. Al igual que con la implementación Clean de árboles AVL, las sentencias *let* fueron reemplazadas por *let* estrictos (*let*!). Como las expresiones declaradas en los *let*'s son computadas siempre, el efecto de esta práctica no introduce un costo adicional, y una vez más se evita la creación de *closures* que hagan referencia a las expresiones locales a ser evaluadas. Al igual que con la implementación Clean de árboles AVL (Secc. 2.3), el efecto de estas modificaciones consiste en una diferencia de performance mínima con respecto a la versión imperativa (ver la Secc. 3.6).

Otra diferencia importante está en la generación de números aleatorios. El compilador Haskell provee funciones predefinidas para generar valores pseudo–aleatorios, mientras que la versión utilizada del compilador Clean no provee una función para obtener números aleatorios que sea fácil de usar en programas que no hagan uso de entrada-salida gráfica. La solución fue implementar una función para generar tales números. El algoritmo implementado es el dado en [L'E88], el cual genera de manera eficiente números pseudo–aleatorios con un período $\approx 2.30584 \times 10^{18}$.

3.4 Treaps Imperativos

La versión procedural de treaps fue desarrollada en C. La implementación se basa en las características de este lenguaje: la estructura interna de los treaps se basa en punteros, y el polimorfismo paramétrico se emula mediante el uso de una función de comparación de tipos genéricos (o sea, `void*`). Las principales operaciones provistas son:

```
/* Crea un Treap con una funcion de comparacion dada */
Treap* newTreap(int (*)(void*, void*));

/* Inserta un elemento en un Treap */
void insertT(Treap*, void*);

/* Borra un elemento dado de un Treap */
void deleteT(Treap*, void*);

/* Verifica si el elemento dado pertenece o no al Treap */
int isElemT(Treap*, void*);
```

Dado que esta tesis centra su estudio en las estructuras de datos funcionales, se omite el estudio en detalle de la versión imperativa. Sólo se dará la implementación de las operaciones de balance, con el objetivo de mostrar la complejidad añadida con respecto a las versiones funcionales. El código completo puede encontrarse en el CD adjunto a este trabajo.

3.4.1 Estructura Interna

La implementación imperativa modela a los treaps mediante tres definiciones de tipos: una para las asociaciones prioridad–elemento, otra para los árboles de binarios de búsqueda, y otra para asociar dichos árboles con la función de comparación de elementos, encargada de proveer la capa de abstracción con respecto al tipo almacenado por el árbol. Como el lenguaje C sí posee efectos laterales, ya no es necesario implementar la generación de números pseudo–aleatorios mediante un *stream* infinito de tales números, tal como se hizo en las implementaciones funcionales. En cambio, las prioridades se obtienen mediante la invocación a un comando apropiado, que retornará un valor aleatorio cada vez que se lo ejecute. Las definiciones de tipo resultantes se listan a continuación:

```
typedef struct TVal
{
    long int priority;
    void* value;
} TVal;

typedef struct BTree
{
    TVal* value;
    struct BTree* left;
    struct BTree* right;
} BTree;

typedef struct Treap
{
    int (*cmp)(void*, void*);
    BTree* tree;
} Treap;
```

3.4.2 Operaciones de Rotación

Las operaciones de rotación imperativas basan su implementación en el uso de punteros. Esto es una ventaja con respecto a las versiones funcionales, debido a que la manipulación directa de punteros y el *update in place* permiten reorganizar un árbol sin crear nuevos nodos. De esta manera se utiliza la memoria de un modo más eficiente que las versiones funcionales, que como se explica en la Secc. 3.2.4, deben crear nuevos nodos para evitar introducir efectos laterales. Sin embargo, el uso óptimo de la memoria hace que la implementación procedural sea mucho más complicada que las versiones funcionales. Esto es una desventaja importante, ya que se expresa una operación esencialmente sencilla de una manera poco intuitiva, y además el uso de punteros es una práctica que como es bien sabido, induce a cometer errores sutiles de difícil detección. Por otra parte, el uso de efectos laterales dificulta la tarea de demostrar formalmente la correctitud de los algoritmos que utilizan estas dos operaciones, dado que ya no es posible recurrir a mecanismos matemáticos sencillos, tales como el principio de inducción completa. A continuación se lista el código para la rotación a derecha (función `rotateRight`). La implementación de la rotación a izquierda es dual, y por lo tanto se omite.

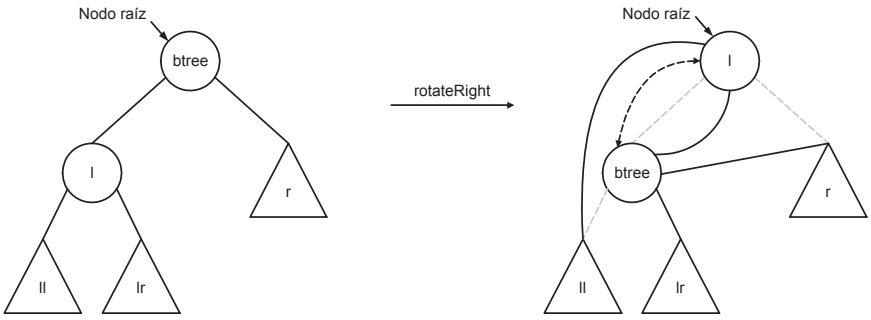

Figura 3.3: Función `rotateRight`

```
void rotateRight(BTree* btree)
{
  TVal* v;
  BTree* l;

  v            = btree->value;
  l            = btree->left;
  btree->value = l->value;
  l->value     = v;
  btree->left  = l->left;
  l->left      = l->right;
  l->right     = btree->right;
  btree->right = l;
}
```

Puede apreciarse cómo la estructura del árbol es modificada mediante el reuso de los distintos componentes del árbol recibido. La Fig. 3.3 ilustra este proceso. Las líneas punteadas negras indican un intercambio entre el contenido de dos nodos, las grises muestran la estructura original del árbol. Notar que el árbol obtenido coincide con que resulta de aplicar las operaciones de rotación funcionales, ilustrado en la Fig. 3.2.

3.5 Correctitud

Para demostrar que las propiedades que definen a un treap permanecen invariantes después de insertar o eliminar un elemento, es necesario probar que el

árbol resultante satisface la condición de búsqueda y el predicado *minHeap*. La primera demostración es sencilla: basta con probar que las operaciones de rotación mantienen la condición de árbol de búsqueda (lo cual también es simple), y hacer inducción sobre la profundidad del árbol. Por lo tanto, se omite este paso y sólo se demostrará que las operaciones de borrado e inserción mantienen *minHeap*. También se demostrará que la profundidad esperada de un treap es $O(\log(n))$, lo que probará la validez de la técnica de insertar elementos en forma aleatoria para obtener un balance cercano al óptimo. Las demostraciones utilizarán la notación ya definida en la Secc. 2.5, donde para todo treap t, d_t denota su profundidad. Además, para todo nodo n, n_p representa su prioridad; y para todo treap t, t_p simboliza la prioridad de su nodo raíz.

3.5.1 Algoritmo de Inserción

Se probará que la función *insertBT* mantiene el predicado *minHeap*. Una vez demostrada esta propiedad, es sencillo probar que *insertT* también la mantiene.

Teorema 3.5.1 *Sea t un árbol binario que satisface* minHeap. *Entonces $t' =$ (*insertBT t v_i*) también satisface* minHeap, *para todo elemento v_i.*

Demostración: Se hará por inducción sobre la altura de t.

Caso Base: El resultado de aplicar *insertBT* sobre un árbol vacío es una hoja, la cual satisface *minHeap*. Por lo tanto, se verifica el caso base.

HI: Para todo nodo v_i y para todo árbol t que satisface *minHeap*, con $d_t < k$, $k \geqslant 1$, *insertBT* t v_i satisface *minHeap*

TI: Para todo nodo v_i y para todo árbol $t = Node\ v\ l\ r$ que satisface *minHeap* con $d_t = k$, $k \geqslant 1$, $t' =$ *insertBT* t v_i también satisface dicho predicado. Para demostrar la tesis inductiva, deben considerarse las tres alternativas de la función *insertBT*:

1. $v = v_i$:
   En este caso, $t' =$ *insertBT* t $v_i = t$. Entonces t' satisface *minHeap*.

2. $v > v_i$:
   Entonces $t' =$ *insertBT* t $v_i = checkLeft(Node\ v\ (insertBT\ l\ v_i)\ r)$.
   Sea $t_i =$ *insertBT* l v_i. Luego, hay dos casos posibles:

(a) $v_p \leqslant t_{ip}$:
Luego, $t' = checkLeft(Node\ v\ t_i\ r) = Node\ v\ t_i\ r$. Por HI, t_i satisface *minHeap*. Además, por hipótesis, $v_p \leqslant t_{ip}$ y $v_p \leqslant r_p$, por lo que t' satisface *minHeap*.

(b) $t_{ip} < v_p$:
Luego, $t' = checkLeft(Node\ v\ t_i\ r) = rotateRight\ (Node\ v\ t_i\ r)$. Como t_i está formado sólo por nodos de l y v_i, y además $t_{ip} < v_p$, la raíz de t_i tiene que ser v_i. Sea $t_i = Node\ v_i\ l_i\ r_i$, entonces,

$$\begin{array}{ll} & rotateRight\ (Node\ v\ t_i\ r) \\ = & \quad \{t_i = Node\ v_i\ l_i\ t_i\} \\ & rotateRight\ (Node\ v\ (Node\ v_i\ l_i\ r_i)\ r) \\ = & \quad \{\text{Definición de } rotateRight, \text{ Secc. 3.2.4}\} \\ & Node\ v_i\ l_i\ (Node\ v\ r_i\ r) \end{array}$$

Por HI, t_i satisface *minHeap*. Luego, l_i también lo satisface y $v_i \leqslant l_{ip}$. Además r_i y r también satisfacen *minHeap*, y como v_i es el único nodo con prioridad menor que v_p, todos los nodos de ambos árboles tienen prioridad mayor que v_p. Entonces $t' = Node\ v_i\ l_i\ (Node\ v\ r_i\ r)$ satisface *minHeap*

3. $v < v_i$:
Este caso es el dual del anterior, por lo que se omite su demostración.

Queda entonces demostrado que la función *insertBT* mantiene el predicado *minHeap* □

3.5.2 Algoritmo de Borrado

Se probará que la función *deleteBT* mantiene la invariancia del predicado *minHeap*. Una vez efectuada esta demostración, resulta sencillo demostrar que *deleteT* también la mantiene. Antes se probará un resultado auxiliar.

Lema 3.5.2 *Sea $t = (Node\ v\ l\ r)$ un árbol binario que satisface minHeap. Entonces $t' = (deleteBT'\ t)$ también lo satisface Además, $v \notin t'$.*

Demostración: El teorema se probará por inducción sobre $d_l \times d_r$.

Caso Base: En este caso $d_l \times d_t = 0$, por lo que l ó r es un árbol vacío (o ambos lo son). Para este último caso, $deleteBT'\ t$ es un árbol vacío, por lo que se cumple el teorema. Si sólo uno de los dos subárboles es vacío, se retorna el otro, el cual satisface *minHeap* porque t lo cumple, y no incluye a

v porque dicho nodo es la raíz de t. Queda así probado el caso base.

HI: Para todo árbol $t = Node\ v\ l\ r$ que satisface *minHeap*, con $d_l \times d_r < k$, $k \geqslant 1$, $deleteBT'\ t$ también lo satisface.

TI: Para todo árbol $t = Node\ v\ l\ r$ que satisface *minHeap*, con $d_l \times d_r = k$, $k \geqslant 1$, $t' = deleteBT'\ t$ también satisface *minHeap*. Para demostrar la tesis inductiva deben analizarse estas dos alternativas:

1. $l_p \leqslant r_p$:
   Sea $l = Node\ v_l\ ll\ lr$, entonces $t = Node\ v\ (Node\ v_l\ ll\ lr)\ r$, y en base a la defínción de $deleteBT'$ (Secc. 3.2.3) puede hacerse la siguiente derivación:

   $$\begin{aligned} & deleteBT'\ t \\ = & \quad \{\text{Definición de } deleteBT', Node\ v_n\ nl\ nr = rotateRight\ t\} \\ & Node\ v_n\ nl\ (deleteBT'\ nr) \\ = & \quad \{rotateRight\ t = Node\ v_l\ ll\ (Node\ v\ lr\ r)\} \\ & Node\ v_l\ ll\ (deleteBT'\ (Node\ v\ lr\ r)) \end{aligned}$$

   Por lo tanto, $t' = Node\ v_l\ ll\ (deleteBT'\ (Node\ v\ lr\ r))$. Como $d_{lr} < d_l$, $d_{lr} \times d_r < d_l \times d_r$. Luego por HI, $deleteBT'\ (Node\ v\ lr\ r)$ satisface *minHeap* y no incluye a v. Dado que v era el único nodo con prioridad menor que v_l (ya que ningún nodo de ll ó lr tiene prioridad menor que v_l, y por hipótesis, $v_{lp} = l_p \leqslant r_p$), t' satisface *minHeap* y además, $v \notin t'$.

2. $l_p > r_p$:
   Este caso es el dual del anterior, por lo que se omite su demostración.

Esto completa la demostración del lema 3.5.2 □

Una vez probado el lema 3.5.2, es sencillo demostrar la correctitud de la función $deleteBT$:

Teorema 3.5.3 *Para todo treap t y todo elemento v_d, $t' = deleteBT\ t\ v$ satisface minHeap y además, $v_d \notin t'$.*

Demostración: Se probará por inducción sobre la altura del árbol.

Caso base: Si t es un árbol vacío, entonces $deleteBT\ t\ v_d$ también está vacío, y el caso base queda probado.

HI: Para todo nodo v_d y todo árbol t que satisfaga *minHeap* tal que $d_t < k$, $k \geqslant 1$, $deleteBT\ t\ v_d$ satisface *minHeap*.

TI: Para todo nodo v_d y todo árbol $t = Node\ v\ l\ r$ que satisfaga *minHeap* tal que $d_t = k$, $k \geqslant 1$, $t' = deleteBT\ t\ v_d$ satisface *minHeap*. Para probar la tesis inductiva deben considerarse las siguientes alternativas:

1. $v = v_d$:
   Entonces $t' = deleteBT'\ t$, y por el lema 3.5.2, t' satisface *minHeap* y no incluye a v_d entre sus nodos.

2. $v > v_d$:
   Luego $t' = Node\ v\ (deleteBT\ l\ v_d)\ r$. Por HI, $deleteBT\ l\ v_d$ satisface *minHeap* y no incluye a v_d, por lo que t' también satisface estas dos propiedades.

3. $v < v_d$:
   Este caso es el dual del anterior y se omite su demostración.

Queda entonces demostrado que la función *deleteBT* mantiene el invariante *minHeap* □

3.5.3 Altura Esperada de un Treap

La complejidad de las operaciones de búsqueda, inserción y borrado sobre un treap están definidas en función de su profundidad máxima. Dado que los treaps reorganizan su estructura basándose en los valores aleatorios de prioridad, no es posible dar una estimación exacta de la profundidad para un treap arbitrario. No obstante, puede modelarse la profundidad de un treap como una variable aleatoria. De este modo puede calcularse su altura esperada.

Se probará que para todo treap t, su altura esperada $O(\log(n))$, siendo n la cantidad de nodos de t. Este resultado es la especialización para treaps de los obtenidos en [DR95, Dev98], donde se demuestra que la profundidad esperada d_t de *todo* árbol binario de búsqueda t construido aleatoriamente es una variable aleatoria tal que $E(d_t) < 4.3 \log(n)$ y $Var(d_t)$ es $O(\log(\log(n))^2)$. Es interesante mencionar que hasta la fecha sólo existe una cota superior para la constante de proporcionalidad de la varianza, siendo la determinación de su valor exacto un problema aún abierto.

Para llevar a cabo la demostración se asumirá que la prioridad es una variable aleatoria con distribución $U[0, 1]$. El motivo para modelar la prioridad como una variable aleatoria continua es forzar a que la probabilidad de que

haya dos nodos con la misma prioridad sea cero, ya que el caso de nodos con prioridades iguales complica la demostración. En la práctica puede suceder que dos nodos tengan la misma prioridad; existen demostraciones más complejas y generales que sí contemplan este caso alcanzando el mismo resultado (por ejemplo, [Dev98]).

Sea x_i el i-ésimo nodo de un treap según su listado inorder. Sea $X(i,k)$ el subconjunto de nodos $\{x_i, x_{i+1}, \ldots, x_k\}$ o el subconjunto $\{x_k, x_{k+1}, \ldots, x_i\}$ según sea $i \leqslant k$ ó $k \leqslant i$, respectivamente. Para todo par (i,k), $X(i,k) = X(k,i)$ y $|X(i,k)| = |k-i|+1$. Se probará el siguiente resultado auxiliar:

Lema 3.5.4 *Para todo treap t y todo par (i,k), x_i es ancestro de x_k (o sea, pertenece al camino que une a x_k con la raíz de t) sii x_i tiene el valor de prioridad más bajo entre todos los nodos pertenecientes a $X(i,k)$.*

Demostración: Se hará por inducción sobre la profundidad de t. Si t está vacío, el lema se cumple por vacuidad. Queda así demostrado el caso base.

HI: Para todo treap t con $d_t < k$, $k \geqslant 1$, y todo par (i,k), x_i es ancestro de x_k sii x_i tiene el valor de prioridad más bajo entre todos los nodos pertenecientes a $X(i,k)$.

TI: La propiedad vale para todo treap t con $d_t = k$, $k \geqslant 1$, y todo par (i,k). Para probar esta afirmación, se deben considerar los siguientes casos:

Si x_i es la raíz de t, es inmediato que x_i es ancestro de x_k. Como todo treap satisface *minHeap*, x_i tiene prioridad más baja que cualquier nodo en treap, y en particular, que cualquier nodo en $X(i,k)$.

Si x_k es la raíz de t, x_i no es ancestro de x_k y tampoco tiene la prioridad más baja en $X(i,k)$ (la tiene x_k).

Sea x_j, $j \neq i$, $j \neq k$, el nodo raíz de t. Si x_j es el mínimo ancestro común de x_i y x_k, se tiene que $i < j < k$, o bien $k < j < i$, por lo que $x_j \in X(i,k)$. En este caso x_i no es ancestro de x_k (porque están en subárboles distintos de t) y no tiene la prioridad más baja en $X(i,k)$ (la tiene x_j).

Finalmente, si el ancestro común x_j de x_i y x_k no es el nodo raíz de t, por HI vale la propiedad para el subárbol cuya raíz es x_j. Además, si x_i está en el camino que une a x_k con x_j, también lo está en el que une a x_k con la raíz de t. Queda entonces demostrado el lema 3.5.4 $\square$

Para analizar la profundidad esperada de un treap t, se define la variable A_k^i de la siguiente manera:

$$A_k^i = \begin{cases} 1, & \text{si } x_i \text{ es ancestro de } x_k \\ 0, & \text{en caso contrario} \end{cases}$$

Por el lema 3.5.4, la probabilidad de que A_k^i sea uno es igual a la probabilidad de que x_i tenga la menor prioridad en $X(i,k)$. Dado que la prioridad tiene distribución uniforme, se tiene que

$$p(A_k^i = 1) = \frac{1}{|X(i,k)|} = \frac{1}{|k-i|+1}$$

Por otra parte, dado que la profundidad de todo nodo $x_k = d(x_k)$ equivale a la cantidad de ancestros propios de x_k, se tiene la siguiente identidad:

$$d(x_k) = \sum_{i=1}^{n} A_k^i - 1$$

Por lo tanto, en base a propiedades algebraicas y de teoría de probabilidades [Dev99], la profundidad esperada de $x_k = E(d(x_k))$ se calcula así:

$$\begin{aligned} E(d(x_k)) &= E\left(\sum_{i=1}^{n} A_k^i - 1\right) \\ &= \sum_{i=1}^{n} E(A_k^i) - 1 \\ &= \sum_{i=1}^{n} p(A_k^i = 1) - 1 \\ &= \sum_{i=1}^{n} \frac{1}{|k-i|+1} - 1 \\ &= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\ &= \sum_{j=2}^{k} \frac{1}{j} + \sum_{j=2}^{n-k+1} \frac{1}{j} \\ &< \ln(k) + \ln(n-k+1) \\ &< 2\ln(n) \end{aligned}$$

Por definición de orden, se tiene como resultado que $E(d(x_k))$ es $O(log(n))$. Esto completa la demostración $\square$

Versión	10^3	10^4	10^5
Haskell	1571.93 Kb	16499.55 Kb	189833.02 Kb
Clean	586.60 Kb	7159.09 Kb	88105.25 Kb
Ratio (%)	37.31%	43.38%	46.41%
Mejora	62.68%	56.61%	53.59%

Tabla 3.1: Consumo de heap de la función *insertT*

3.6 Análisis de Eficiencia

El análisis de eficiencia sobre treaps se basa en los mismos casos de prueba que los utilizados para los árboles AVL. Esto es, se analizó el consumo de memoria y el tiempo de ejecución resultantes de insertar 10^3, 10^4 y 10^5 elementos generados pseudo–aleatoriamente en un treap vacío, seguido del cálculo de la altura del árbol resultante (ver la Secc. 2.6). Las condiciones bajo las que fue efectuado el análisis también son idénticas. Esto es, los casos de prueba fueron ejecutados bajo el sistema operativo Linux, y el rendimiento de la versión Haskell fue analizado en base a los reportes producidos por el *profiler* provisto por dicho compilador. Para la versión Clean, los tiempos de ejecución fueron obtenidos bajo Linux, y el consumo de memoria fue estudiado según los resultados producidos por el *profiler* provisto por el compilador Clean para Win32. Como el rendimiento bajo ambas plataformas es similar, los resultados obtenidos son aplicables a Linux.

3.6.1 Eficiencia Espacial

Las pruebas efectuadas muestran que el rendimiento espacial de la versión Clean es mucho mejor que el presentado por la versión Haskell. Esto se debe principalmente a la propiedad de Clean de controlar mediante anotaciones de *strictness* los fragmentos de código y componentes de la estructura de datos que requieren ser estrictos.

La Tabla 3.1 muestra los resultados obtenidos. Puede verse la diferencia existente entre la memoria consumida por ambas implementaciones. El uso de heap por parte de la versión Clean representa entre el 37% y el 46% del consumido por la versión Haskell, lo que equivale a una mejora que varía entre el 62% y el 53%, decreciente según aumenta el tamaño de la entrada.

La Fig. 3.4 muestra un gráfico comparativo para los valores listados en la Tabla 3.1. Dado que el eje x crece según una progresión aritmética de factor 10, se utilizan coordenadas logarítmicas de modo tal que el eje y crezca en la misma proporción y se refleje la verdadera magnitud de la curvas obtenidas.

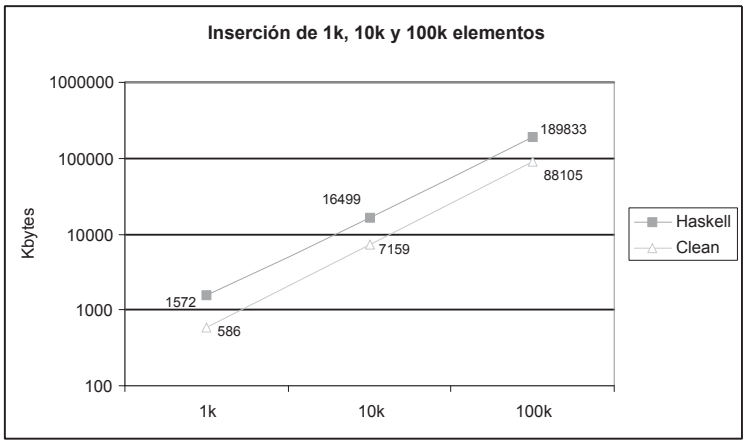

Figura 3.4: Uso de heap de las versiones funcionales de *insert*

$f_S(i)$	$i = 3$	$i = 4$	$i = 5$
Haskell	1.572	1.649	1.898
Clean	0.586	0.716	0.881

Tabla 3.2: Constantes de proporcionalidad de complejidad espacial

La complejidad espacial de un treap con n elementos es $O(n)$, ya que un nodo almacena un único elemento. Para analizar la relación entre los valores obtenidos y esta fórmula, se estudiarán los resultados de la función $f_S = \frac{S(i)}{10^i}$, $i \in \{3, 4, 5\}$, donde $S(i)$ es la memoria resultante de insertar en un treap vacío 10^i elementos. Como f_S computa aproximaciones a las distintas constantes de proporcionalidad, debería comportarse aproximadamente como una función constante para cada implementación. Los valores tomados por f_S se muestran en la Tabla 3.2. Notar que las constantes de proporcionalidad de Clean son menores que la mitad de las obtenidas para Haskell. Si bien los valores están próximos entre sí, presentan una tendencia levemente creciente. El análisis de los consumos reportados por las herramientas de *profiling* revela que la posible razón es la existencia de referencias a nodos del treap al momento de ejecutarse la rutinas de manejo automático de memoria, por lo que no son removidos del heap.

3.6.2 Eficiencia Temporal

Dado que la eficiencia temporal está directamente relacionada con la espacial, el tiempo de ejecución de la implementación Clean es mejor que el de la versión Haskell. La Tabla 3.3 lista los tiempos de ejecución obtenidos. Con

Versión	10^3	10^4	10^5
Haskell	20 ms	440 ms	5400 ms
Clean	10 ms	130 ms	1920 ms
C	7 ms	90 ms	970 ms

Tabla 3.3: Tiempos de ejecución de la función *insertT*

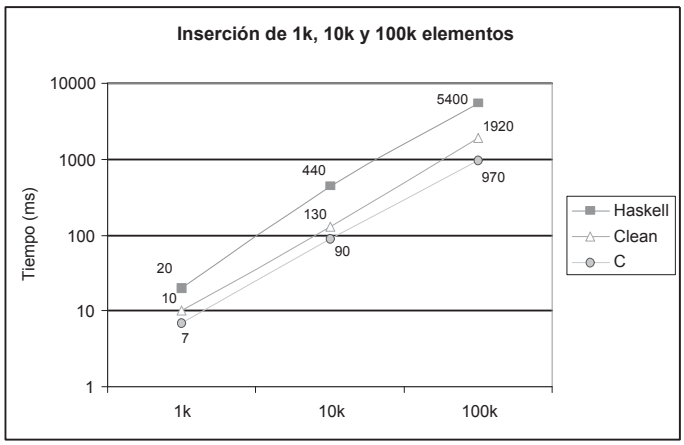

Figura 3.5: Tiempo de ejecución de la función *insertT*

respecto a los tiempos de ejecución de la versión C, la implementación Haskell tarda sólo 2.85, 4.88 y 5.56 veces más para las entradas de 10^3, 10^4 y 10^5 elementos, respectivamente. Por otra parte, la versión Clean es todavía más eficiente, tardando sólo 1.42, 1.44 y 1.97 veces más que la versión C para dichas entradas.

La Fig. 3.5 muestra un gráfico comparativo de los tiempos de ejecución obtenidos, utilizando también coordenadas logarítmicas para las ordenadas, dado que el eje x crece en progresión aritmética. Notar la cercanía de la curva correspondiente a la versión Clean con respecto a la obtenida por la implementación C.

La complejidad de efectuar n inserciones en un treap es $O(n \log(n))$. Para analizar la relación entre esta fórmula y los tiempos de ejecución obtenidos se efectuará el mismo procedimiento que en la Secc. 3.6.1. Esto es, se estudiará el comportamiento de la función $f_T = 10^3 \frac{T(i)}{10^i \log(10^i)}$, $i \in \{3, 4, 5\}$, donde $T(i)$ es el tiempo que toma efectuar 10^i inserciones en un treap vacío. El producto por 10^3 es para evitar los errores que podrían introducirse por falta de precisión del sistema de numeración de punto flotante. Dado que f_T computa el valor aproximado de la constante de proporcionalidad, f_T debería comportarse como una función constante para cada implementación.

$f_T(i)$	$i = 3$	$i = 4$	$i = 5$
Haskell	2.006	3.311	3.251
Clean	1.003	0.978	1.155
C	0.702	0.677	0.583

Tabla 3.4: Constantes de proporcionalidad de complejidad temporal

Los valores tomados por dicha función se listan en la Tabla 3.4. Puede verse que f_T tiene poca variación, de acuerdo a lo esperado. Las únicas excepciones se presentan con la versión Haskell, para la entrada de tamaño 10^3, y con la versión Clean para la entrada de 10^5 elementos. En el primer caso, la constante es mucho menor que las dos restantes porque esta es la única entrada para la cual no se realiza *garbage collection*. De este modo, los resultados para 10^4 y 10^5 elementos son mayores debido al tiempo adicional que introduce el manejo automático de memoria. Para Clean, la justificación es similar: la entrada de tamaño 10^5 es la única para la que se efectúa *garbage collection*, lo que hace que la constante de proporcionalidad aumente. De hecho, el compilador reporta 490 ms de *garbage collection*, lo que da un tiempo neto de ejecución $T(i), i = 5$, de 1430 ms. El valor de f_T para dicho tiempo es 0.861, lo que se aproxima mucho más a los valores obtenidos para las pruebas restantes.

3.6.3 Conclusiones

Los resultados obtenidos a partir de las pruebas de rendimiento espacial y temporal muestran que la implementación Clean es mucho más eficiente que la versión Haskell. Los motivos principales que justifican esta diferencia son la implementación sumamente eficiente que presenta Clean, y la gran flexibilidad de las *strictness annotations* provistas por dicho lenguaje para especificar qué expresiones deben ser computadas según un orden de evaluación estricto y cuáles no. Una ventaja adicional de las anotaciones de *strictness* es que no tienen un impacto significativo en la claridad del código resultante luego de ser aplicadas. A su vez, es importante poder determinar con precisión qué fragmentos de código necesitan ser optimizados. Por lo tanto, las técnicas de *profiling* son otro factor importante para obtener implementaciones funcionales puras eficientes.

Debe notarse que disponer de anotaciones de *strictness* no necesariamente implica eliminar completamente el uso de la *lazy evaluation* para obtener eficiencia. Los versión Clean de treaps es un buen ejemplo de esta afirmación, ya que en su estructura interna coexisten un árbol binario estricto (que

modela al treap en sí) y una lista infinita (por lo tanto, *lazy*) de números generados al azar. Sin embargo, esta implementación es casi tan eficiente como la versión C.

La Secc. 3.6.2 muestra que las rutinas de *garbage collection* pueden tener una incidencia importante sobre los tiempos de ejecución resultantes, ya que aumentan las constantes de proporcionalidad que los relacionan con la complejidad del algoritmo. Este es un argumento más a favor de los mecanismos de especificación de *strictness* como forma de controlar indirectamente el consumo de memoria de un programa funcional puro. También debe notarse que el lenguaje C no posee la capacidad de manejar memoria automáticamente. Como los casos de pruebas efectuados sólo insertan nodos (o sea, no liberan memoria explícitamente), el análisis de eficiencia tiende a favorecer a C, ya que no insume tiempo en esta tarea. Esto hace aún más valiosos los resultados obtenidos para las versiones Haskell y Clean.

Los valores listados en esta sección concuerdan con las conclusiones obtenidas para árboles AVL en la Secc. 2.6.3: mediante la utilización de técnicas de *profiling* y *strictness annotations* es posible obtener tiempos de ejecución comparables a los de un lenguaje procedural incluso para entradas de datos relativamente grandes.

Capítulo 4

Árboles de van Emde Boas

Hasta ahora, este trabajo ha estudiado sólo estructuras de datos funcionales puras, utilizando como caso de estudio tipos particulares de árboles balanceados. En este capítulo se estudiarán los árboles de van Emde Boas [vEBKZ77] (VEBTrees, para abreviar), los cuales son una estructura de datos netamente imperativa que provee una modelización eficiente de conjuntos cuyos elementos soportan aritmética entera y pertenecen a un universo finito. Los VEBTrees se utilizan comúnmente para modelizar colas de prioridad. Se proveen cuatro operaciones básicas: *insert*, *delete*, *pred* y *succ*. Su implementación más eficiente se basa en arreglos mutables, motivo por el cual no se trata de una estructura de datos funcional pura.

Los VEBTrees reúnen características deseables de diversas estructuras de datos más estudiadas. Por ejemplo, para un universo $U = \{1\,..\,u\}$, con $U \subset \mathbb{Z}$, todas las operaciones provistas se resuelven en $O(\log\log(u))$ pasos. Para $u = 2^{64}$, esto significa efectuar 4 iteraciones para efectuar una inserción. Esta complejidad es comparable a la de las *hash tables*, que si bien permiten insertar o eliminar en tiempo $O(1)$, dicho tiempo es amortizado debido al manejo de colisiones. A su vez, un VEBTree preserva el orden de los elementos almacenados: obtener el primer o el último elemento requiere $O(1)$ pasos. Por lo tanto, los VEBTrees proveen la misma funcionalidad que los heaps, pero con mayor eficiencia. Finalmente, se verá en la Secc. 4.1 que la búsqueda de un elemento también puede resolverse en $O(\log\log(u))$ pasos.

La desventaja principal de los VEBTrees la constituye su capacidad fija: un VEBTree puede almacenar a lo sumo los elementos del universo finito que está modelando. Por otra parte, restringen los elementos a almacenar a aquellos que soportan aritmética entera. No obstante, puede asociarse a cada cada tipo a almacenar un valor numérico, por lo que el problema puede solucionarse al costo de modificar el diseño de los tipos de datos a ser insertados. Sin embargo, para facilitar el estudio y sin pérdida de generalidad,

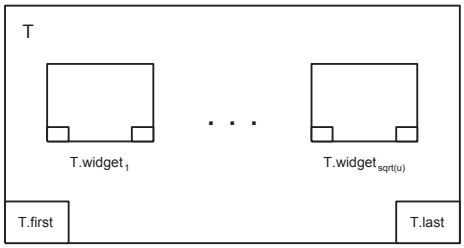

Figura 4.1: Estructura de un VEBTree

en lo que resta del capítulo se asumirá que todo universo U es de la forma $U = \{1\,..\,u\}, U \subset \mathbb{Z}$.

Este capítulo está organizado de la siguiente manera. La Secc. 4.1 provee una introducción a los VEBTrees, dando su estructura interna y el pseudo–código de sus cuatro operaciones. Las Secc. 4.2 y 4.3 discuten las implementaciones Haskell y Clean, respectivamente. Dado que Haskell modela los arreglos mutables (y las demás operaciones que introducen efectos laterales) mediante mónadas, mientras que Clean lo hace mediante *unique types*, ambas implementaciones son completamente diferentes, y también los resultados de performance obtenidos. La Secc. 4.4 discute la versión C de VEBTrees. Finalmente, la Secc. 4.5 analiza y compara la eficiencia de las tres implementaciones dadas anteriormente.

4.1 Introducción a los VEBTrees

Un VEBTree de orden u es un árbol que modela a un universo finito $U = \{1\,..\,u\}$ (o sea, sólo puede almacenar valores menores o iguales que u), cuyos nodos consisten en una estructura llamada *widget*. Todo *widget* (si no está vacío) almacena el primer y último elemento de un árbol, y contiene una colección de $\sqrt{u}$ *widgets* (o sea, subárboles) de orden $\sqrt{u}$. Intuitivamente, un elemento puede ser el primero, el último o un valor intermedio; en este último caso, debe estar almacenado en uno de los *widgets* de la colección, o sea, en un subárbol. Para elegir el subárbol apropiado, se divide el elemento por $\lceil \sqrt{u} \rceil$; el cociente indica el número de *widget*, y el resto, el elemento a almacenar en él. Puesto que los *widgets* son de orden $\sqrt{u}$, no es posible almacenar el número original. La Fig. 4.1 muestra la estructura de un VEBTree.

Sea T un VEBTree de orden u. Sea S el conjunto de elementos almacenados en T, $S \subseteq U$. Entonces, la estructura de T está definida por siguiente invariante:

$$\begin{aligned} VEB \equiv\ & T = \emptyset \vee \\ & (min\{S\} = T.first \ \wedge \\ & max\{S\} = T.last \ \wedge \\ & \forall\ v \in S, T.first < v < T.last : v = a\lceil \sqrt{u}\ \rceil + b \ \wedge \ b \in T.widget_a) \end{aligned}$$

donde $\emptyset$ simboliza un VEBTree vacío, $T.first$ y $T.last$ denotan el primer y último elemento del VEBTree T, y $T.widget_a$ se refiere al a-ésimo *widget* de v. El invariante VEB define una manera simple de obtener el *widget* correspondiente a un elemento dado. Además, los *widgets* de un VEBTree de orden u son de orden $\sqrt{u}$. Por lo tanto, es posible descomponer cualquiera de las cuatro operaciones provistas para VEBTrees de orden u en un subproblema de orden $\sqrt{u}$ sobre el *widget* apropiado. De este modo, el tiempo de ejecución de estas operaciones puede definirse mediante la siguiente ecuación de recurrencia:

$$T(u) = \begin{cases} c_1 & , \text{ si } u = 1 \\ c_2 + T(\sqrt{u}) & , \text{ en caso contrario} \end{cases} \tag{4.1}$$

Esta ecuación define la complejidad de las operaciones de inserción, borrado, sucesor y predecesor como $O(\log\log(u))$. Intuitivamente, esto puede verse ya que hacen falta $\log(u)$ bits para codificar a u en binario; dado que $\sqrt{u}$ necesita la mitad de esa cantidad, resulta que el número inicial decrece por mitades. Por lo tanto, toma $\log\log(u)$ pasos reducir este número a un bit. La demostración formal es la siguiente.

Teorema 4.1.1 *La complejidad de un problema cuyo tiempo de ejecución está definido por la ecuación 4.1 es* $O(\log\log(n))$

Demostración: Dado que se trabaja con aritmética entera, se puede asumir que el paso de $T(2)$ a $T(1)$ se hace en un tiempo constante k. Por otra parte, puede verse que $T(u) = ic_2 + T(\sqrt[2^i]{u})$. Dado este patrón, puede despejarse i de la siguiente manera: $u^{2^{-i}} = 2 \ \equiv \ 2^{-i}\log(u) = 1 \ \equiv \ \log\log(u) = i$. Resulta que $T(u) = \log\log(u)c_2 + k + c_1 < (c_2 + c_1 + k)\log\log(u)$, por lo tanto, $T(u)$ es $O(\log\log(u))$ $\square$

Las cuatro operaciones provistas deben mantener el invariante VEB, y a la vez, su tiempo de ejecución tiene que estar dado por la ecuación 4.1. Estas dos condiciones hacen que la implementación de dichas operaciones no sea trivial. Por motivos de claridad, las operaciones se estudiarán de manera progresiva. A modo introductorio, se dará una versión inicial de las funciones *insert* y *succ*. Si bien ambas funciones mantienen VEB, *succ* no

tiene complejidad $O(\log\log(u))$. Por lo tanto, se presentará una variante mejorada de la función *succ* que sí cumple con ambas condiciones, pero que fuerza a modificar la implementación de *insert*. La operación *delete* es la más complicada, por lo que se la estudiará separadamente. El estudio de la operación *pred* se omite, dado que es la dual de *succ*. Notar que las funciones *succ* y *pred* proveen una manera trivial de definir la función de búsqueda *isElem* con complejidad $O(\log\log(u))$, constituída por la ecuación $isElem\ t\ e \equiv pred\ t\ (e+1) = e \equiv succ\ t\ (e-1) = e$.

4.1.1 Funciones *insert* y *succ*: Versión Inicial

Las operaciones se llevarán a cabo sobre un VEBTree T de orden u y por claridad, se escribirá $\sqrt{u}$ en lugar de $\lceil\sqrt{u}\ \rceil$. Para presentar los algoritmos se usará la siguiente notación. La expresión $a \leftarrow b$ denota una asignación donde a pasa a tener el valor de b. A su vez, $a \rightarrow e$ es un sinónimo de "sea a un valor igual a e". Por ejemplo, $v \rightarrow a\sqrt{u} + b$ es una manera conveniente de denotar el cociente y módulo de la operación $v/\sqrt{u}$. Dichos valores (a y b en este caso) pueden ser utilizados en el resto del pseudo–código. Dos o más expresiones pueden ser secuenciadas mediante el símbolo ",". Finalmente, *succ* puede retornar un valor nulo, representado mediante el símbolo $\bullet$.

Función *insert*

Para mantener *VEB*, la función *insert* debe distinguir tres alternativas diferentes. Si el VEBTree T donde se insertará un elemento v está vacío, v pasa a ser a la vez el primer último elemento de T; por lo tanto, los campos $T.first$ y $T.last$ deben ser inicializados con dicho valor. Si T tiene un único elemento, entonces v pasará a ser el primero o el último, según sea menor o mayor que el valor contenido en el árbol. Si T tiene más de dos elementos, hay tres casos posibles. Si v es menor que $T.first$, $T.first$ debe ser insertado en el *widget* correspondiente y reemplazado por v. Del mismo modo, si el valor a insertar es mayor que $T.last$, $T.last$ debe insertarse en el *widget* apropiado y reemplazarse por v. Si v corresponde a un valor intermedio, debe insertarse recursivamente en el *widget* adecuado.

function *insert*(T,v)
  if $T = \emptyset$ **then**
    $T.first \leftarrow T.last \leftarrow v$
  if $T.first = T.last$ **then**
    $T.first \leftarrow min\{v, T.first\},\ T.last \leftarrow max\{v, T.last\}$
  if $v < T.first$ **then**

$T.first \to a\sqrt{u} + b,\ insert(T.widget_a, b),\ T.first \leftarrow v$
if $v > T.last$ **then**
$T.last \to a\sqrt{u} + b,\ insert(T.widget_a, b),\ T.last \leftarrow v$
if $T.first < v < T.last$ **then**
$v \to a\sqrt{u} + b,\ insert(T.widget_a, b)$
return T
end function

Informalmente, se puede apreciar que el algoritmo de inserción mantiene *VEB*. Además, es sencillo comprobar que su complejidad es $O(\log\log(u))$. Las primeras dos condiciones se ejecutan en tiempo constante, mientras que las restantes aplican recursivamente el algoritmo de inserción sobre una entrada de tamaño $\sqrt{u}$ y son excluyentes. Por lo tanto, el tiempo de ejecución de *insert* está dado por la ecuación 4.1, que corresponde a la complejidad buscada.

Función *succ*

Para esta función existen cuatro alternativas posibles. Si el árbol T donde se buscará el sucesor está vacío, se debe retornar $\bullet$, o sea, el elemento nulo. Si el elemento v cuyo sucesor se desea obtener es menor que $T.first$, debe retornarse $T.first$. Si v es mayor que $T.last$, se debe retornar el valor nulo. En caso que v sea un valor intermedio, debe obtenerse el *widget* correspondiente. Si está vacío o su último elemento es menor v, entonces el sucesor no se encuentra en él. Por lo tanto, se debe obtener el próximo *widget* no vacío, y si existe, retornar su primer elemento, o $T.last$ en caso contrario (ya que no hay sucesores en los *widgets*, pero v es menor $T.last$). Si el sucesor sí se halla en el *widget* correspondiente, debe invocarse recursivamente la función *succ* sobre él. El invariante *VEB* define que para todo elemento v que no es ni el primero ni el último de un VEBTree de orden u se almacena el valor b en el *widget* w-ésimo, donde $v = w\sqrt{u} + b$. Por lo tanto, para recuperar el valor original v a partir del valor b en el *widget* w debe retornarse $w\sqrt{u} + b$.

function *succ*(T,v)
if $T = \emptyset$ **then return** $\bullet$
if $v < T.first$ **then return** $T.first$
if $v > T.last$ **then return** $\bullet$

$v \to a\sqrt{u} + b$
if $T.widget_a = \emptyset$ **then**
$w \leftarrow$ índice del próximo *widget* no vacío, o $\bullet$ si no existe

```
    if w ≠ •
      then return w√u + T.widget_w.first
      else return T.last
  else
    if b < T.widget_a.last
      then return a√u + succ(T.widget_a, b)
      else w ← índice del próximo widget no vacío, o • si no existe
           if w ≠ •
             then return w√u + T.widget_w.first
             else return T.last
end function
```

Esta función no modifica la estructura del árbol, por lo que mantiene *VEB*. En cuanto al análisis de la complejidad temporal, las tres primeras alternativas se ejecutan en tiempo constante. En el resto de la función se efectúan tres operaciones distintas. Por un lado se tienen asignaciones, que se ejecutan en tiempo constante, y una invocación recursiva a *succ* sobre un *widget* que representa a un árbol con tamaño $\sqrt{u}$. El problema está provocado por la operación de búsqueda del próximo *widget* no vacío. Dado que T tiene $\sqrt{u}$ *widgets*, esta operación ejecuta en el peor caso en $O(\sqrt{u})$ pasos. Resulta entonces que este algoritmo no tiene la complejidad buscada.

4.1.2 Funciones *insert* y *succ*: Versión Final

La función *succ* se compone de alternativas disjuntas que se ejecutan en $O(\log\log(u))$ pasos, salvo las que deben encontrar el próximo *widget* no vacío. Por lo tanto, para que el tiempo de ejecución de *succ* pase a ser $O(\log\log(u))$ es necesario que la búsqueda del próximo *widget* tome también a lo sumo ese tiempo. Esto puede lograrse a partir de las siguiente observación clave: *el conjunto de widgets no vacíos de un VEBTree T de orden u puede modelizarse mediante un nuevo VEBTree de orden $\sqrt{u}$*. Esto es, el número $i, i \in 0\,..\,\sqrt{u}$ pertenecerá a este nuevo VEBTree sii $T.widget_i \neq \emptyset$. De esta manera, encontrar el próximo *widget* no vacío reduce a la función *succ* sobre este VEBTree de orden $\sqrt{u}$. Luego, el tiempo de ejecución pasa a estar definido según la ecuación 4.1, con lo que *succ* pasa a tener la complejidad deseada. Este nuevo *widget* se llamará *resumen*, y para un árbol T se denotará mediante $T.res$.

El *widget* resumen debe reflejar en todo momento el estado de los *widgets* restantes. Por lo tanto, todo VEBTree T debe mantener un invariante adicional:

$$RES \;\equiv\; T = \emptyset \;\vee\; \forall\, a \in 0\,..\,\sqrt{u} :\; T.widget_a \neq \emptyset \Leftrightarrow a \in T.res$$

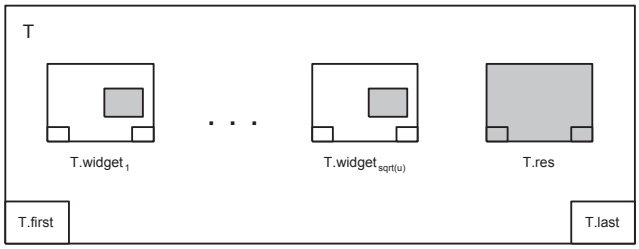

Figura 4.2: Estructura de un VEBTree: versión mejorada

Dicho invariante deberá ser mantenido por las funciones que modifican la estructura del árbol, o sea, *insert* y *delete*. La Fig. 4.2 muestra la nueva estructura de un VEBTree inducida por el nuevo *widget* resumen.

Función *insert*

Para que *insert* mantenga *RES* además de *VEB*, es necesario que luego de ser efectuada una inserción se agregue el *widget* modificado al resumen, si es que anteriormente estaba vacío. Para esto, la función *insert* debe retornar además del nuevo árbol un booleano que indique si el *widget* donde se realizó la inserción estaba vacío o no. En caso afirmativo, se debe efectuar una inserción adicional en el resumen. Si bien ahora podría parecer que se efectúan dos inserciones por llamada recursiva, se puede observar que sólo es necesario insertar en el resumen si la inserción se realizó en un *widget* vacío; dado que esto toma tiempo constante entonces sólo una de las llamadas recursivas se ejecuta en su totalidad. En consecuencia, la complejidad del algoritmo no se ve afectada.

function *insert*(T,v)
  if $T = \emptyset$ **then**
    $T.first \leftarrow T.last \leftarrow v$
    return(T, **True**)
  if $T.first = T.last$ **then**
    $T.first \leftarrow min\{v, T.first\},\ T.last \leftarrow max\{v, T.last\}$
  if $v < T.first$ **then**
    $T.first \rightarrow a\sqrt{u} + b$
    $(T, empty) \leftarrow insert(T.widget_a, b),\ T.first \leftarrow v$
    if *empty* **then** $insert(T.res, a)$
  if $v > T.last$ **then**
    $T.last \rightarrow a\sqrt{u} + b$
    $(T, empty) \leftarrow insert(T.widget_a, b),\ T.last \leftarrow v$

```
    if empty then insert(T.res, a)
  if T.first < v < T.last then
    v → a√u + b
    (T, empty) ← insert(T.widget_a, b)
    if empty then insert(T.res, a)
  return (T, False)
end function
```

Función *succ*

La implementación mejorada de la función *succ* debe obtener el próximo *widget* no vacío mediante una invocación recursiva sobre el resumen, la cual corresponde a ejecutar el algoritmo sobre una entrada de tamaño $\sqrt{u}$. El tiempo de ejecución de *succ* queda entonces tal como se lo definió en la ecuación 4.1, y por lo tanto, su complejidad pasa a ser $O(\log\log(u))$. Al igual que en la versión anterior, se mantienen sobre el árbol los invariantes *VEB* y *RES*, dado que la función *succ* no modifica su estructura.

```
function succ(T,v)
  if T = ∅ then return •
  if v < T.first then return T.first
  if v > T.last then return •

  v → a√u + b
  if T.widget_a = ∅ then
    w ← succ(T.res, a)
    if w ≠ •
      then return w√u + T.widget_w.first
      else return T.last
  else
    if b < T.widget_a.last
      then return a√u + succ(T.widget_a, b)
      else w ← succ(T.res, a)
           if w ≠ •
             then return w√u + T.widget_w.first
             else return T.last
end function
```

4.1.3 La función *delete*

El algoritmo de borrado es el más complejo, ya que debe distinguir entre cinco casos posibles. Si el árbol T sobre el que se ejecutará la operación está

vacío, se debe retornar T. Si T tiene un único elemento igual al valor v a borrar, hay que retornar el árbol vacío. Si $T.first < v < T.last$, se debe ejecutar recursivamente el algoritmo de borrado sobre el *widget* apropiado. Sea a el índice de dicho *widget*. En caso de que $T.widget_a$ quede vacío después de borrar a v, se debe eliminar el valor a de $T.res$, de modo que se mantenga *RES*. Si bien parecería que se están ejecutando dos operaciones de borrado, si se ejecuta *delete* $T.res$ a entonces $T.widget_a$ quedó vacío, luego tenía un solo elemento y la primera invocación a *delete* se hizo en tiempo constante. Si $v = T.first$, se debe obtener el primer elemento del primer *widget* no vacío, eliminarlo de dicho *widget* y utilizarlo para reemplazar a $T.first$, preservando así *VEB*. Nuevamente, el resumen permite realizar esta operación en $O(\log\log(u))$ pasos. Si $v = T.last$, se debe encontrar el último elemento del último *widget*, eliminarlo y usarlo para reemplazar $T.last$. El pseudo–código completo del algoritmo es el siguiente:

function *delete*(T,v)
  if $T = \emptyset$ **then return** T
  if $v = T.first \ \wedge \ v = T.last$ **then return** $\emptyset$
  if $T.first < v < T.last$ **then**
    $v \rightarrow a\sqrt{u} + b$
    $delete(T.widget_a, b)$
    if $T.widget_a = \emptyset$ **then** $delete(T.res, a)$
  if $v = T.first$ **then**
    $a \leftarrow T.res.first$
    $b \leftarrow T.widget_a.first$
    $T.first \leftarrow a\sqrt{u} + b$
    $delete(T.widget_a, b)$
    if $T.widget_a = \emptyset$ **then** $delete(T.res, a)$
  if $v = T.last$ **then**
    $a \leftarrow T.res.last$
    $b \leftarrow T.widget_a.last$
    $T.last \leftarrow a\sqrt{u} + b$
    $delete(T.widget_a, b)$
    if $T.widget_a = \emptyset$ **then** $delete(T.res, a)$
  return T
end function

Al igual que con las demás operaciones, cada alternativa del algoritmo de borrado se ejecuta en tiempo constante o recursivamente sobre una entrada de tamaño $\sqrt{u}$, con lo que su tiempo de ejecución queda definido por la ecuación 4.1, y su complejidad por $O(\log\log(u))$.

4.2 VEBTrees: Versión Haskell

Debido a que uno de los objetivos de este trabajo es explorar y comparar distintas alternativas para la implementación de estructuras de datos en un entorno funcional puro, se presentan dos implementaciones Haskell de VEB-Trees: una modela el conjunto de *widgets* mediante arreglos no mutables, mientras que la otra se vale de arreglos que soportan *update in place* (mutables). Dado que la implementación de las operaciones sobre VEBTrees fue discutida en la Secc. 4.1, para ambas versiones sólo se estudiará su estructura interna y la implementación de la función *insert*, la cual conforma fragmento significativo del código que bastará para exponer las cuestiones de diseño e implementación que debieron ser consideradas.

4.2.1 VEBTrees no Mutables

La implementación de los VEBTrees no mutables se basa en este tipo de arreglos, por lo tanto primero se explicará cómo los mismos son modelados por Haskell.

Arreglos no Mutables en Haskell

El tipo utilizado por Haskell para modelar arreglos no mutables es $Array\ a\ b$, donde a es el tipo utilizado como índice. Las operaciones principales provistas por esta estructura de datos son las siguientes:

$$
\begin{array}{ll}
array & :: Ix\ a \Rightarrow (a, a) \rightarrow [(a, b)] \rightarrow Array\ a\ b \\
(!) & :: Ix\ a \Rightarrow Array\ a\ b \rightarrow a \rightarrow b \\
(//) & :: Ix\ a \Rightarrow Array\ a\ b \rightarrow [(a, b)] \rightarrow Array\ a\ b
\end{array}
$$

La función *array* crea un nuevo arreglo dado un par que denota los índices inferior y superior del arreglo a crear, y una lista de pares índice-valor que representa su contenido inicial. El operador (!) permite seleccionar un elemento de un arreglo dado su índice, y el operador (//) actualiza un arreglo dada una lista de pares índice-valor a reemplazar. El contexto *Ix* expresa que el tipo a debe implementar la funcionalidad impuesta por Haskell para los elementos utilizados como índices (en particular, a debe ser un conjunto ordenado, comparable y enumerable).

El problema que presenta esta estructura de datos (conocido como el problema del *update in place*) es que, como todas las estructuras no monádicas, no puede actualizarse sin generar una copia. Para ver por qué, considérese el siguiente ejemplo:

$$f \;=\; \textbf{let}\; a \;=\; \mathit{array}\;(0,0)\;[(0,0)]\;\textbf{in}\;(a\,!\,0, a\,//\,[(0,1)]) \tag{4.2}$$

Si a fuese actualizado sin retornar una copia, el resultado de extraer la primera componente de la tupla retornada por el ejemplo dependerá de si fue evaluada o no la segunda. Por lo tanto, el resultado depende del orden de evaluación, el cual no puede ser establecido a priori por ser Haskell un lenguaje lazy. La necesidad de generar una copia fuerza a que la operación de *update* se ejecute en tiempo $O(n)$, al contrario de lo que sucede con los arreglos procedurales, donde se ejecuta en tiempo constante. El problema del *update in place* se discute más detalladamente en el Ap. A.

Esta característica de los arreglos no mutables hace que estructuras de datos cuya implementación se basa en dichos arreglos no puedan ser implementadas mediante un lenguaje funcional puro con la misma eficiencia que con un lenguaje imperativo. En el caso particular de los VEBTrees, las operaciones *insert* y *delete* tendrán una complejidad proporcional a la longitud del arreglo de *widgets*, o sea, para un VEBTree de orden u se ejecutarán en $O(\sqrt{u})$ pasos, en lugar de $O(\log\log(n))$.

Implementación

Utilizando esta abstracción de arreglos, la signatura de las operaciones provistas por el tipo abstracto VEBTree son las siguientes:

$$\begin{aligned}
&\mathit{insert} &&:: (\mathit{Integral}\ u,\ \mathit{Ix}\ u) \Rightarrow \mathit{VEBTree}\ u \to u \to \mathit{VEBTree}\ u\\
&\mathit{delete} &&:: (\mathit{Integral}\ u,\ \mathit{Ix}\ u) \Rightarrow \mathit{VEBTree}\ u \to u \to \mathit{VEBTree}\ u\\
&\mathit{succ} &&:: (\mathit{Integral}\ u,\ \mathit{Ix}\ u) \Rightarrow \mathit{VEBTree}\ u \to u \to \mathit{Maybe}\ u\\
&\mathit{pred} &&:: (\mathit{Integral}\ u,\ \mathit{Ix}\ u) \Rightarrow \mathit{VEBTree}\ u \to u \to \mathit{Maybe}\ u
\end{aligned}$$

El contexto *Integral* expresa que los elementos de un VEBTree deben soportar aritmética entera. Además, los elementos se utilizan para calcular el índice

del *widget* apropiado, por lo que también deben ser instancia del contexto *Ix*. El tipo *Maybe u* modela una computación que puede fallar de manera controlada, o bien retornar un valor de tipo *u* si tiene éxito (ver el Ap. A).

La estructura interna de un VEBTree se define de modo tal que contemple las distintas alternativas que deben analizar los algoritmos presentados en la Secc. 4.1. Esto es, un VEBTree puede estar vacío, tener un único elemento, dos elementos, o más de dos, lo que implica la existencia de elementos intermedios que estarán almacenados en sus *widgets* correspondientes y de un resumen indicando cuáles de ellos están vacíos y cuáles no. Para todas las alternativas, un VEBTree conoce su orden, dado por el primer parámetro de todos sus constructores. El orden es necesario para poder determinar cuantos *widgets* contendrá el árbol (un VEBTree de orden u tiene $\sqrt{u}$ *widgets*). En base a estas consideraciones, un VEBTree basado en arreglos mutables está definido por el siguiente tipo algebraico:

```
data VEBTree u
  = EmptyVEB u                  — VEBTree vacío
  | OneVEB u u                  — VEBTree con un único elemento
  | TwoVEB u u u                — VEBTree con dos elementos
  | ManyVEB u u u               — primer y último elemento
    (Array u (VEBTree u))       — √u widgets
    (VEBTree u)                 — resumen
```

Dada esta definición de VEBTrees, el pseudo–código del algoritmo de inserción dado en la Secc. 4.1.2 queda implementado tal como se muestra en Fig. 4.3. Por razones de claridad, se han omitido los contextos de las signaturas de tipo. Las dos primeras alternativas de la función *insert* implementan trivialmente la inserción sobre un árbol vacío y uno unitario, respectivamente. La inserción sobre un árbol con dos elementos reduce a una inserción recursiva sobre un nuevo árbol donde el conjunto de *widgets* y el resumen ya han sido inicializados. Para este último caso se verifica si el nuevo elemento pasa a ser el primero, el último, o uno intermedio del árbol. Para cada uno de estos casos se invoca a la función *ins* con los parámetros adecuados. Esta función devuelve un árbol donde el segundo parámetro es su primer elemento, el tercero el último, y el primero ha sido insertado en la posición intermedia adecuada.

Además de los arreglos no mutables, existe otra fuente de ineficiencia: la copias que hacen las funciones *insert* e *insertWasEmpty* del VEBTree donde

```
insert :: VEBTree u → u → VEBTree u
insert t i = (fst . insertWasEmpty) t i

insertWasEmpty :: VEBTree u → u → (VEBTree u, Bool)
insertWasEmpty (EmptyVEB ru) i = (OneVEB ru i, True)
insertWasEmpty t@(OneVEB ru u) i =
 case compare i u of
   LT → (TwoVEB ru i u, False)
   EQ → (t, False)
   GT → (TwoVEB ru u i, False)
insertWasEmpty t@(TwoVEB ru f l) i
   | i == f || i == l = (t, False)
   | otherwise =
     let
       rru = sqr ru
       det = array (0, ru − 1) [(j, emptyVEB rru) | j ← [0..ru − 1]]
       res = emptyVEB rru
     in
     insertWasEmpty (ManyVEB ru f l det res) i
insertWasEmpty t@(ManyVEB ru f l det res) i =
 case compare i f of
   EQ → (t, False)
   LT → ins f i l
   GT → case compare i l of
          EQ → (t, False)
          GT → ins l f i
          LT → ins i f l
   where
     ins i f l =
      let
        (a, b)          = i `divMod` ru
        (w, wasEmpty)   = insertWasEmpty (det ! a) b
        det'            = det // [(a, w)]
        res'            = if wasEmpty then insert res a else res
      in
      (ManyVEB ru f l det' res', False)
```

Figura 4.3: Función *insert*: versión no mutable

se lleva a cabo la inserción. No es posible retornar el mismo VEBTree actualizado por el mismo motivo que no posible hacer *update in place* sobre un arreglo no mutable (ver el ejemplo 4.2). La próxima sección muestra cómo Haskell permite solucionar este problema.

4.2.2 VEBTrees Mutables

Para que los VEBTrees funcionales implementen las operaciones *insert* y *delete* en tiempo $O(\log\log(u))$, es necesario modelar el conjunto de *widgets* mediante un arreglo mutable.

Arreglos Mutables en Haskell

Haskell modela arreglos mutables mediante una mónada de estados [Wad92b, PJW93, Lau93, LPJ94, LPJ95]. Dicha mónada, llamada *ST*, permite modelar computaciones que pueden actualizar un estado en tiempo constante, tal como el definido por la semántica operacional de los lenguajes imperativos. Si un arreglo pertenece a dicho estado mutable, podrá ser actualizado en tiempo constante y el problema del *update in place* quedará resuelto.

Para comprender por qué la mónada *ST* permite resolver este problema, es necesario examinar cómo está definida y cómo se utiliza. La mónada está modelada por un tipo *ST s a*, donde *s* es el estado mutable, y *a* el resultado retornado por una computación que puede modificar el estado *in place*. Haskell provee una sintaxis especial para facilitar la codificación de programas monádicos llamada *do-notation* (ver el Ap. A). La idea es permitir la secuenciación de acciones monádicas con una sintaxis pseudo–imperativa. Dicha secuencia se escribe de la siguiente manera:

do
  [{]
    $action_1$ [;]
    ...
    $action_n$ [;]
  [}]

Una acción monádica puede retornar un valor, o sólo modificar el estado. En el primer caso, el resultado se referencia mediante la notación $res \leftarrow action$. Si una acción monádica tiene esta forma, la variable *res* puede utilizarse en cualquier otra que suceda a la que produjo el valor.

Un arreglo mutable se modela mediante el tipo abstracto *STArray s i e*, donde *s* es la componente mutable (el estado) donde "reside" el arreglo, *i* es el tipo de los índices del arreglo, y *e* el de los elementos a ser almacenados.

Al igual que con los arreglos no mutables, el tipo i debe ser instancia de la clase *Ix*. Entre otras, este tipo de arreglos presenta estas operaciones:

$$
\begin{aligned}
&\mathit{newSTArray} &&:: \mathit{Ix}\ i \Rightarrow (i,i) \to e \to \mathit{ST}\ s\ (\mathit{STArray}\ s\ i\ e)\\
&\mathit{readSTArray} &&:: \mathit{Ix}\ i \Rightarrow \mathit{STArray}\ s\ i\ e \to i \to \mathit{ST}\ s\ e\\
&\mathit{writeSTArray} &&:: \mathit{Ix}\ i \Rightarrow \mathit{STArray}\ s\ i\ e \to i \to e \to \mathit{ST}\ s\ ()
\end{aligned}
$$

La función *newSTArray* crea un arreglo mutable con los índices inferior y superior dados en el primer parámetro, inicializado con el valor dado por el segundo. La función *readSTArray* retorna un elemento dado un índice, y *writeSTArray* modifica *in place* (sin hacer una copia) un arreglo dado. Esta función sólo modifica el estado que modela al arreglo, sin retornar ningún valor de utilidad (el tipo () modela un conjunto unitario cuyo elemento consiste en una 0-upla). Mediante las construcciones presentadas, el ejemplo 4.2 que causaba el problema del *update in place* se escribe así:

```
do
  arr ← newSTArray (0,0) 0    — arreglo con un solo elemento = 0
  val ← readSTArray arr 0     — lee la posición 0
  writeSTArray arr 0 1        — actualiza in place el arreglo
  return (val, arr)           — retorna el valor y el arreglo
```

Notar que la mónada *ST* fuerza de manera transparente una manipulación secuencial del estado. Esto hace que el orden de evaluación esté definido, por lo que el problema del efecto lateral introducido por el *update in place* desaparece. Sin embargo, las mónadas presentan algunas desventajas: impiden razonar de modo ecuacional sobre un programa, añaden complejidad al código, y puede ser difícil (o imposible) combinarlas con otras mónadas [KW92], lo que limita la reusabilidad del código monádico.

Implementación

Dado que los VEBTree mutables tienen una implementación monádica, el tipo de sus operaciones varía en el resultado que retornan, que será del tipo *ST*. La signatura de dichas operaciones pasa a ser:

```
insert ::
   (Integral u, Ix u) ⇒ VEBTree s u → u → ST s (VEBTree s u)
delete ::
   (Integral u, Ix u) ⇒ VEBTree s u → u → ST s (VEBTree s u)
succ   ::
   (Integral u, Ix u) ⇒ VEBTree s u → u → ST s (Maybe u)
pred   ::
   (Integral u, Ix u) ⇒ VEBTree s u → u → ST s (Maybe u)
```

La nueva definición del tipo abstracto que modela a los VEBTrees se obtiene a partir de la dada en la sección anterior para la versión no mutable, reemplazando el tipo *Array a b* por un arreglo mutable de tipo *STArray s a b*. Como los VEBTrees incluyen un arreglo mutable, su tipo debe estar parametrizado con el estado *s* que modela a dicho arreglo, además del tipo de los elementos a almacenar. De este modo, el tipo algebraico utilizado para definir a los VEBTrees es ahora:

```
data VEBTree s u
  = EmptyVEB u                      — VEBTree vacío
  | OneVEB u u                      — VEBTree unitario
  | TwoVEB u u u                    — VEBTree con dos elementos
  | ManyVEB u u u                   — primer y último elemento
    (STArray s u (VEBTree s u))     — arreglo mutable de √u widgets
    (VEBTree s u)                   — resumen
```

Obviando las diferencias sintácticas introducidas por el uso de la *do-notation*, la versión mutable de la función *insert* está estructurada de la misma manera que la no mutable, expuesta en la sección anterior. La Fig. 4.4 muestra el código completo de dicha función. Nuevamente por motivos de claridad, se han omitido los contextos de las declaraciones de tipo.

La implementación mutable de VEBTrees presenta diferencias notables de performance con respecto a la versión funcional pura. Dicha diferencia se debe fundamentalmente a la eficiencia de los arreglos mutables contra la de los funcionales puros. Los detalles acerca de esta diferencia de rendimiento se dan en la Secc. 4.5.

```
insert :: VEBTree s u → u → ST s (VEBTree s u)
insert t i = do { (t', _) ← insertWasEmpty ti; return t' }


insertWasEmpty :: VEBTree s u → u → ST s (VEBTree s u, Bool)
insertWasEmpty (EmptyVEB ru) i = return (OneVEB ru i, True)
insertWasEmpty t@(OneVEB ru u) i =
case compare i u of
  LT → return (TwoVEB ru i u, False)
  EQ → return (t, False)
  GT → return (TwoVEB ru u i, False)
insertWasEmpty t@(TwoVEB ru f l) i
  | i == f || i == l = return (t, False)
  | otherwise =
     let rru = sqr ru in
      do
         evt ← emptyVEB rru
         res ← emptyVEB rru
         det ← newSTArray (0, ru − 1) evt
         insertWasEmpty (ManyVEB ru f l det res) i
insertWasEmpty t@(ManyVEB ru f l det res) i =
case compare i f of
  EQ → return (t, False)
  LT → ins f i l
  GT → case compare i l of
          EQ → return (t, False)
          GT → ins l f i
          LT → ins i f l
  where
    ins i f l =
     let (a, b) = i `divMod` ru in
      do
         w ← readSTArray det a
         (w', wasEmpty) ← insertWasEmpty w b
         writeSTArray det a w'
         res' ← if wasEmpty then insert res a else return res
         return (ManyVEB ru f l det res', False)
```

Figura 4.4: Función *insert*: versión mutable

4.3 VEBTrees: Versión Clean

La versión Clean de VEBTrees se basa en la implementación que provee Clean de arreglos mutables. Existen dos motivos fundamentales para utilizar Clean además de Haskell para implementar VEBTrees:

1. El diseño del lenguaje Clean aplica el siguiente precepto: *un lenguaje funcional eficiente debe proveer arreglos mutables eficientes*. Los arreglos mutables provistos por Clean están altamente optimizados [Gro97] y son mucho más eficientes que los implementados por Haskell. Por lo tanto, es de esperarse que la implementación Clean de VEBTrees sea más eficiente que la obtenida con Haskell.

2. Clean modela arreglos mutables mediante *unique types* (discutidos en el Ap. A), lo que permite explorar alternativas diferentes de las mónadas para implementar programas funcionales puros que incluyan efectos imperativos.

Una expresión con tipo *unique* sólo puede tener a lo sumo una referencia a lo largo de su tiempo de vida dentro de un programa. La verificación de esta propiedad es efectuada por la fase de chequeo de tipos en tiempo de compilación. Es responsabilidad del programador usar un tipo *unique* de modo tal que no se viole la propiedad de unicidad, o sea, que el programa tenga un tipo válido según el algoritmo de chequeo de tipos. Esta manipulación explícita pone a los *unique types* en desventaja con respecto a las mónadas, que también permiten modelar y manipular un estado mutable, pero de un modo transparente para el programador. Esto hace que las mónadas sean mucho más fáciles de utilizar que los *unique types*. No obstante, los *unique types* no requieren la introducción de una notación especializada, no impiden el razonamiento ecuacional, ni fuerzan un estilo de programación que pueda comprometer la reutilización de código.

Arreglos Mutables en Clean

Sea a un tipo Clean cualquiera, entonces $\{a\}$ es el tipo de los arreglos con elementos de tipo a. Este tipo de arreglos es no mutable, o sea, una actualización sobre un arreglo de tipo $\{a\}$ forzará una copia del arreglo. Para que un arreglo sea mutable debe ser también *unique*. Dicho atributo se denota mediante el símbolo $*$, esto es, $*\{a\}$ es un arreglo mutable que puede ser actualizado en tiempo constante.

Las operaciones sobre arreglos provistas por Clean pueden clasificarse en dos categorías: las que operan exclusivamente sobre arreglos mutables, y las

que funcionan sobre ambos tipos de arreglos. Para esta última categoría (y en general, todas las funciones que reciben argumentos que pueden ser o no *unique*), se introduce el operador (.) como mecanismo para denotar *posible* unicidad. Sea $f :: .a \rightarrow .b$; si f se invoca con un parámetro *unique*, entonces todo tipo precedido por (.) se considera *unique*, por lo que el valor retornado también lo es. Igualmente, si se invoca a f con un argumento no *unique*, tampoco lo será $.b$, por lo que el resultado de la aplicación no es *unique*. Las operaciones principales provistas por Clean para ambos tipos de arreglos son las siguientes:

$$
\begin{array}{ll}
\mathit{createArray} & :: !\mathit{Int} \;\; e \rightarrow .\{e\} \\
\mathit{select} & :: !.\{e\} \;\; !\mathit{Int} \rightarrow e \\
\mathit{uselect} & :: !*\{e\} \;\; !\mathit{Int} \rightarrow (e, !*\{e\}) \\
\mathit{update} & :: !*\{.e\} \;\; !\mathit{Int} \;\; .e \rightarrow .\{.e\} \\
\mathit{replace} & :: !*\{.e\} \;\; !\mathit{Int} \;\; .e \rightarrow (.e, !*\{.e\})
\end{array}
$$

Las signaturas de tipo son ligeramente distintas a las de Haskell: el símbolo ($\rightarrow$) es único y separa los parámetros (separados entre sí mediante espacios) del resultado. Notar también el uso el operador de *strictness* (!), que fuerza la evaluación del argumento correspondiente antes de invocar una función (también se lo usa para denotar resultados estrictos).

La función *createArray* crea un arreglo, dado su tamaño inicial y un elemento de inicialización; el arreglo resultante es posiblemente *unique*: su unicidad será resuelta por el compilador en función de cómo se lo utilice en el resto del programa. La función *select* permite seleccionar un elemento de un arreglo posiblemente *unique*. Es de destacar que esta función anula la unicidad del arreglo recibido, ya que agrega una referencia: si el arreglo tenía una sola, ahora pasa a tener dos, ya que *select* también lo referencia. En efecto, *select a i* referencia al arreglo a, por lo tanto, cualquier referencia posterior no puede tratar al arreglo a como *unique*. Para poder obtener elementos de un arreglo sin violar su unicidad debe utilizarse *uselect*, la cual retorna además del elemento deseado, una *nueva* versión del arreglo, sin referenciar. De este modo, es posible seguir utilizando como *unique* este nuevo arreglo. Notar que el elemento del arreglo retornado por *uselect* (y también por *select*) no puede ser *unique*, dado que ya está referenciado por el arreglo donde se almacena. La función *update* permite actualizar *in place* una posición dada de un arreglo mutable. La unicidad del arreglo resultante depende de la

unicidad del nuevo elemento: si hay más de una referencia a algún elemento de un arreglo, no puede ser considerado *unique*. La función *replace* permite seleccionar un elemento de un arreglo mutable sin alterar su posible atributo de unicidad. Esto se logra dando un elemento que reemplazará en el arreglo al ya seleccionado. De este modo, no aumenta la cantidad de referencias al elemento original, ya que dejó de ser referenciado por el arreglo que lo contenía. Esta función se utiliza en lugar de *select* y *uselect* para seleccionar elementos *unique* preservando dicho atributo.

Para ilustrar el uso de estas funciones se implementará el ejemplo 4.2, utilizado en la Secc. 4.2.1 como ejemplo del problema del *update in place*.

---

let
  a = *createArray* 1 0 — arreglo con un solo elemento = 0
  (v, a_1) = *uselect* a 0 — obtiene el elemento
  a_2 = *update* a_1 0 1 — actualiza *in place* el *nuevo* arreglo
in
(v, a_2) — retorna el viejo valor y el arreglo actualizado

Notar que al igual que con la solución monádica implementada por Haskell, los *unique types* fuerzan un orden de evaluación sobre las expresiones que componen el programa.

Utilizar arreglos mutables sin violar la propiedad de unicidad puede ser una tarea complicada, sobre todo si los elementos del arreglo también son *unique*. Ejemplos de estructuras con elementos que también son *unique* son los VEBTrees, que modelan el conjunto de *widgets* mediante un arreglo mutable de VEBTrees *unique*; y las matrices mutables, que se modelan mediante arreglos mutables cuyos elementos son también arreglos mutables.

Implementación

Dado que un VEBTree contendrá un arreglo *unique* de VEBTrees, el tipo que modela a estos árboles también debe ser *unique*. A su vez, las operaciones sobre VEBTrees deben retornar un nuevo VEBTree para preservar la unicidad del árbol. A continuación se detalla la signaturas de las operaciones sobre el tipo abstracto VEBTree. Sus nombres incluyen el sufijo *VEB* porque la biblioteca standard de Clean define una función llamada *insert*. El contexto *VebElem* expresa que el tipo a debe modelar un conjunto ordenado, y soportar aritmética entera.

```
insertVEB :: !*(VEBTree a) !a -> !*(VEBTree a) | VebElem a
deleteVEB :: !*(VEBTree a) !a -> !*(VEBTree a) | VebElem a
succVEB   ::
  !*(VEBTree a) !a -> !*(!*VEBTree a, !.Maybe a) | VebElem a
predVEB   ::
  !*(VEBTree a) !a -> !*(!*VEBTree a, !.Maybe a) | VebElem a
```

El tipo algebraico que modela a un VEBTree es similar al utilizado en la versión Haskell, salvo por el operador de posible unicidad para el arreglo y las anotaciones de *strictness*.

```
:: VEBTree a = EmptyVEB !a          — VEBTree vacío
             | OneVEB !a !a         — VEBTree unitario
             | TwoVEB !a !a !a      — dos elementos
             | ManyVEB !a !a !a     — primer y último elemento
               !.{VEBTree a}        — √u widgets
               !(VEBTree a)         — resumen
```

Al igual que en la sección anterior, por motivos de claridad sólo se presentará el algoritmo de inserción. Dado que Clean fuerza a preservar explícitamente la unicidad de los arreglos mutables, la implementación presenta diferencias importantes con respecto a la implementación Haskell. No obstante, el código sigue respetando la estructura del algoritmo dado en la Secc. 4.1.2. La inserción para los casos base de un árbol vacío o unitario es trivial, y para un árbol con dos elementos reduce a inicializar los componentes restantes (el conjunto de *widgets* y el resumen), e insertar recursivamente sobre el árbol resultante. Notar que el arreglo que modela a los *widets* (*det*) sea crea a partir de una lista de *ru* VEBTrees vacíos, la cual se genera de manera *lazy* mediante la función *repeatn*. El caso final se resuelve mediante la función *ins*, cuyo comportamiento es el mismo que en la versión Haskell.

El listado completo del algoritmo se incluye en la Fig. 4.5. Por motivos de claridad, se omiten los contextos *VebElem* de las signaturas de tipo de las funciones que componen el algoritmo.

```
insertVEB :: !*(VEBTree a) !a -> !*(VEBTree a)
insertVEB t i = fst (insertWasEmpty t i)

insertWasEmpty :: !*(VEBTree a) !a -> !*(!*VEBTree a, !*Bool)
insertWasEmpty (EmptyVEB ru) i = (OneVEB ru i, True)
insertWasEmpty t =: (OneVEB ru u) i
  | i < u   = (TwoVEB ru i u, False)
  | i == u  = (t, False)
  | i > u   = (TwoVEB ru u i, False)
insertWasEmpty t =: (TwoVEB ru f l) i
  | i == f || i == l = (t, False)
  = let!
      rru = sqr ru
      det = { emptyVEB u \\ u <- repeatn (toInt ru) rru }
      res = emptyVEB rru
    in insertWasEmpty (ManyVEB ru f l det res) i
insertWasEmpty (ManyVEB ru f l det res) i
  | i == f = (ManyVEB ru f l det res, False)
  | i < f  = ins f i l
  | i > f
      | i == l = (ManyVEB ru f l det res, False)
      | i > l  = ins l f i
               = ins i f l
  where
    ins i f l = let! a   = i / ru
                     a_i = toInt a
                     b   = i mod ru
                     (w, det)   = replace det a_i (emptyVEB zero)
                     (w, empty) = insertWasEmpty w b
                     det        = update det a_i w
                in if (not empty) (ManyVEB ru f l det res, False)
                                  let! res = insertVEB res a
                                   in (ManyVEB ru f l det res, False)
```

Figura 4.5: Función *insert*: versión Clean

4.4 VEBTrees: Versión C

La implementación C, como sucede con las tres versiones funcionales presentadas anteriormente, es una traducción relativamente directa del pseudo–código dado en la Secc. 4.1. Sin embargo, dadas las características del lenguaje tales como manejo de punteros y administración manual de memoria, es la más complicada y extensa de todas las presentadas.

Debido a que el tipo de los elementos de un VEBTree está restringido a conjuntos finitos cuyos elementos deben soportar aritmética entera, no se hace uso de punteros a función para emular polimorfismo paramétrico tal como se hizo en las Secc. 2.4 y 3.4. En cambio, un VEBTree sólo puede almacenar elementos de tipo `unsigned long`. El tipo VEBTree se modela en C contemplando los cuatro estados posibles de un VEBTree: vacío, unitario, con dos elementos, o con más de dos. Luego, la estructura interna resulta en la siguiente definición de tipo:

```
enum VST
{
  EMPTY_VEB = 0, ONE_VEB, TWO_VEB, MANY_VEB
};

typedef struct VEBTree
{
  VST state;
  unsigned long ru;
  unsigned long f;
  unsigned long l;
  struct VEBTree** det;
  struct VEBTree*  res;
} VEBTree;
```

El campo `state` actúa como tag, diferenciando en cuál de los cuatro estados posibles se encuentra la estructura. El resto de los campos modela los valores usuales: el orden de árbol, el primer elemento, el último, el arreglo de *widgets* y el resumen. Según el estado del VEBTree, el valor de estos campos debe ser ignorado. La signatura de las operaciones para la versión C se detallan a continuación:

```
/* Crea un VEBTree vacio con el orden dado */
VEBTree* topVEB(unsigned long);

/* Inserta un elemento en un VEBTree */
void insert(VEBTree*, unsigned long);

/* Borra de un VEBTree un elemento */
void delete(VEBTree*, unsigned long);

/* Retorna el sucesor de un elemento (-1 si no existe) */
unsigned long succ(VEBTree*, unsigned long);

/* Retorna el predecesor de un elemento (-1 si no existe) */
unsigned long pred(VEBTree*, unsigned long);
```

Los algoritmos C que implementan las operaciones del tipo abstracto VEBTree son altamente complicados y no contribuyen con el estudio de la implementación de estructuras de datos en un entorno funcional puro. Por lo tanto, se omite el estudio de dichos algoritmos; la implementación completa puede encontrarse en el CD adjunto a este trabajo.

La importancia de esta versión radica en que provee una implementación imperativa contra la cual comparar el rendimiento de las versiones funcionales presentadas con anterioridad en este capítulo. Esta tarea se lleva a cabo en la Secc. 4.5, donde se compara y analiza el rendimiento de las versiones funcionales contra esta implementación.

4.5 Análisis de Eficiencia

El análisis de eficiencia se basa en la inserción de 10^3, 10^4 y 10^5 elementos en orden pseudo–aleatorio. Como la versión Haskell no mutable es *lazy*, las inserciones no serán computadas si no se realiza algún cálculo sobre todos los elementos del árbol resultante. Por lo tanto, luego de insertar los elementos correspondientes se invoca a una función que calcula su altura. Esta función recorre todos los nodos del árbol, lo que asegura la inserción de todos los elementos. La mónada *ST* es estricta en el estado, por lo que no es necesario la función auxiliar para forzar la inserción en la versión Haskell mutable.

Tampoco lo es para la versión Clean, debido a que las anotaciones de *strictness* hacen a la implementación estricta. Sin embargo, se aplica el mismo test para facilitar la comparación entre todas las versiones.

Todas los pruebas se llevaron a cabo bajo el sistema operativo Linux, utilizando un heap que oscila entre los 30 y 60 Mbytes. Para las versiones Haskell, los tiempos de ejecución y datos sobre el consumo de memoria fueron obtenidos mediante las herramientas de *profiling* que provee el lenguaje. Para la versión Clean, los tiempos se obtuvieron mediante los reportes producidos por el programa una vez finalizada su ejecución, mientras que el consumo de memoria fue analizado bajo Win32. Esto se debe a que Clean no provee herramientas de *profiling* para Linux. Sin embargo, dado que los tiempos de ejecución para Win32 son similares a los de Linux, los resultados son extrapolables a dicha plataforma.

4.5.1 Eficiencia Espacial

El análisis de eficiencia espacial se hará sólo para las versiones funcionales, dado que el *profiler* C no provee datos sobre el consumo de memoria. Existen diferencias de performance notables entre las tres versiones funcionales. Estas se deben a dos factores fundamentales:

- La diferencia entre la performance de los distintos tipos de arreglos provistos por Haskell y Clean, explicados respectivamente en las Secc. 4.2.2 y 4.3

- La mejora de consumo de memoria introducida por los fragmentos de código estricto. La mónada ST es estricta en el estado, por lo que los arreglos mutables de Haskell son estrictos. Esto no sucede con las arreglos no mutables, lo que ocasiona una diferencia importante entre el uso de memoria de las versiones Haskell. La anotaciones de *strictness* provistas por Clean también provocan una diferencia importante a favor de la implementación en dicho lenguaje.

En general, los lenguajes *lazy* hacen uso de la memoria de un modo distinto (y por lo general menos eficiente) del utilizado por los lenguajes estrictos. Esto se debe a que un lenguaje *lazy* suspende la evaluación de expresiones hasta que son necesarias, con lo que el heap tiende a superpoblarse con referencias a computaciones a ser evaluadas (*closures*). La diferencia entre las versiones Haskell con arreglos estrictos y *lazy* puede apreciarse en las Fig. 4.6 y 4.7. Estos diagramas muestran la evolución del uso de memoria a lo largo de la inserción de 10^4 elementos por parte de ambas versiones. Las figuras fueron

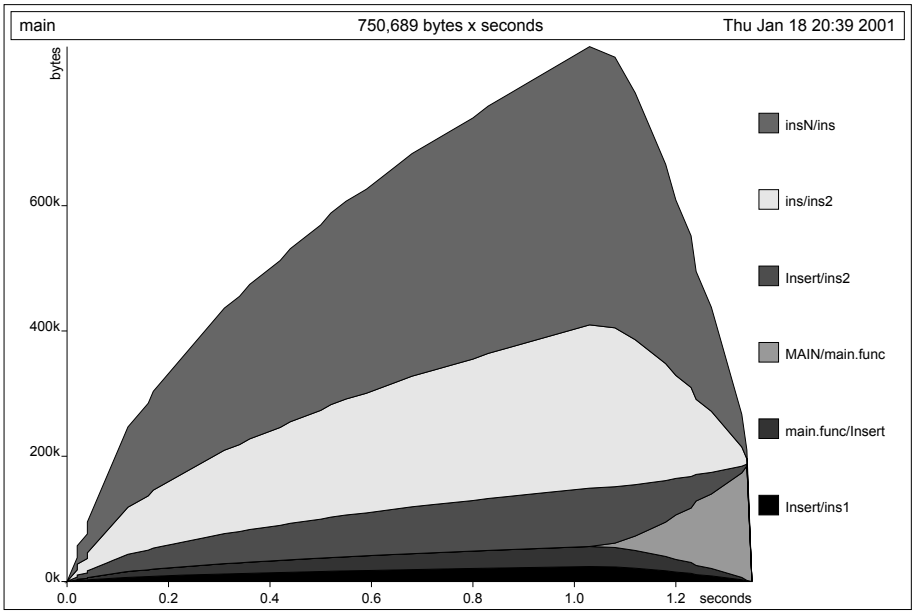

Figura 4.6: Heap profile para la función *insert*, versión monádica

obtenidas mediante el *heap profiler* provisto por el lenguaje. Para las demás entradas (10^3 y 10^5) los gráficos son similares.
Los símbolos ins0, ins1, ins2 e insN encontrados en ambas figuras corresponden a las distintas alternativas del algoritmo de inserción: árbol vacío, unitario, con dos elementos y con más de dos, respectivamente. El símbolo ins corresponde a la función auxiliar *ins*. Puede apreciarse que el consumo de la versión monádica evoluciona de acuerdo a lo que se esperaría de un programa imperativo, creciendo conforme se efectúan las 10^4 inserciones. La versión con arreglos *lazy* almacena inicialmente los *closures* correspondientes a las inserciones a ser computadas. A medida que la función que calcula la altura del árbol va forzando su computación, los *closures* van siendo reemplazados por los resultados obtenidos, por lo que disminuye gradualmente el consumo de heap. La diferencia entre el área de las dos curvas muestra que la eficiencia espacial de la implementación *lazy* es peor que la de la estricta. Por este mismo motivo, la versión Clean hace uso de las anotaciones de *strictness* para lograr un rendimiento espacial mucho mejor al que se obtendría en el caso de una implementación totalmente *lazy*.

La Tabla 4.1 muestra el consumo de memoria resultante del test de inserción para las distintas implementaciones de VEBTrees. La inserción de 10^5 elementos falla para la versión Haskell no mutable. Esto sucede porque el consumo de memoria de esta versión supera a la disponible para ejecutar el algoritmo. Puede verse que el consumo de la versión monádica mejora al de

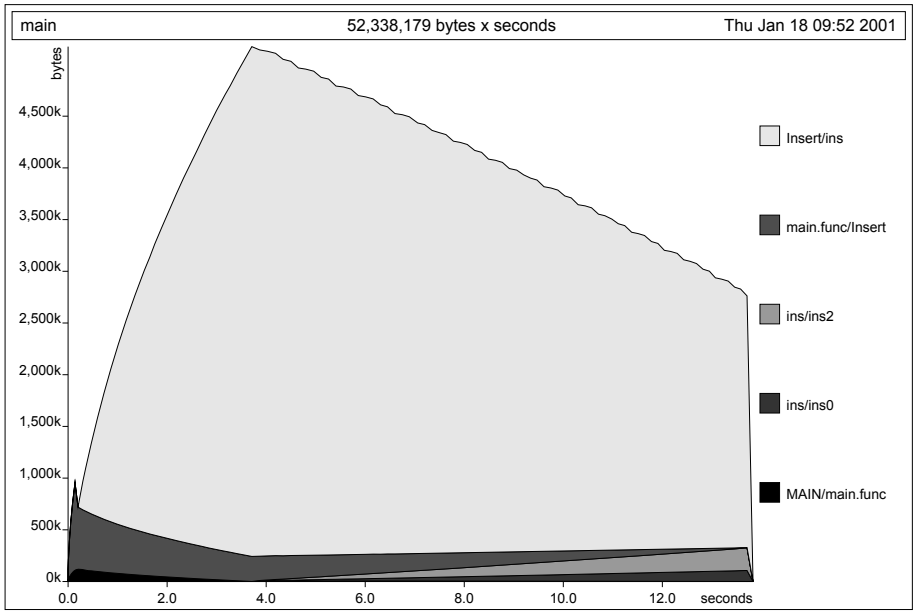

Figura 4.7: Heap profile para la función *insert*, versión no mutable

Versión	10^3	10^4	10^5
Haskell no mutable	2252.70 Kb	30625.18 Kb	—
Haskell mutable	1376.31 Kb	14933.68 Kb	154704.62 Kb
Clean	317.27 Kb	4015.52 Kb	38889.55 Kb

Tabla 4.1: Consumo de heap de la función *insert*

la versión no mutable en un factor que ronda el 50%. A su vez, la eficiencia espacial de la versión Clean mejora a la de la versión Haskell mutable en un factor aproximado del 75%. La Fig. 4.8 muestra un gráfico comparativo para dichos valores. Debido a que el eje x crece en una progresión aritmética de factor 10, se utilizan coordenadas logarítmicas donde el eje y crece en la misma proporción. De este modo se mantiene la verdadera forma de la curvas obtenidas.

En [vEBKZ77] se demuestra que la complejidad espacial de un VEBTree de orden u es $O(u \log\log(u))$. Sea $S(i)$ la cantidad de memoria consumida por un VEBTree de orden $u = 10^i$ para una entrada de u elementos. Para analizar la relación entre los resultados obtenidos y esta ecuación se estudiará a la función $f_S(i) = \frac{S(i)}{10^i \log\log(10^i)}$, $i \in \{3, 4, 5\}$. Dado que esta función computa una aproximación a la constante de proporcionalidad de la fórmula de complejidad espacial, f_S debería ser aproximadamente constante para cada implementación. Los valores que toma f_S para cada versión se listan

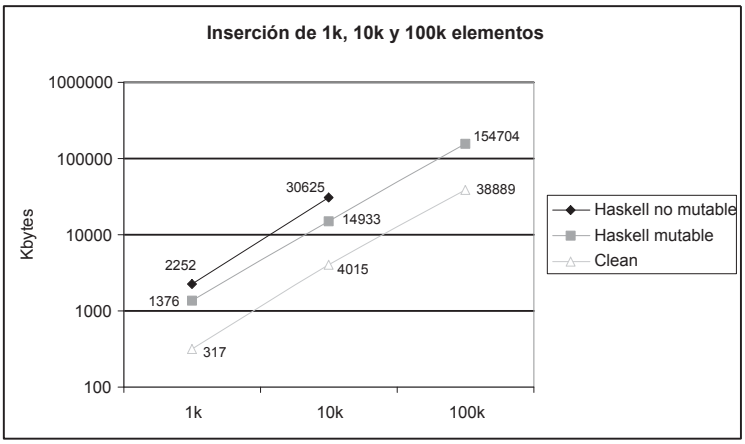

Figura 4.8: Gráfico del uso de heap dado en la Tabla 4.1

$f_S(i)$	$i = 3$	$i = 4$	$i = 5$
Haskell no mutable	0.679	0.820	∞
Haskell mutable	0.414	0.400	0.381
Clean	0.095	0.107	0.096

Tabla 4.2: Constantes de proporcionalidad de complejidad espacial

en la Tabla 4.2. Para la versión Haskell no mutable existe un crecimiento considerable al pasar de 10^3 a 10^4 elementos, el cual puede atribuirse a la naturaleza *lazy* de su implementación, que tiende a crear nodos para los valores intermedios utilizados para efectuar las inserciones. Las versiones restantes exhiben un comportamiento con poca variación. Notar que la constante de la implementación Clean es mucho menor que la de las versiones Haskell.

4.5.2 Eficiencia Temporal

Analizando las implementaciones de la función *insert*, puede apreciarse que las alternativas para el árbol vacío, unitario y con dos elementos constituyen los casos base, que se resuelven en tiempo constante. Luego, el tiempo de ejecución de la función *insert* está defindo por el último caso, cuya única alternativa no trivial está implementada por la función *ins*. Dicha función efectúa una lectura del arreglo de *widgets* y una inserción recursiva, lo que implica una actualización del arreglo. Por lo tanto, la eficiencia de las operaciones sobre arreglos es el otro factor esencial en la eficiencia temporal de *insert*. Esta observación permite explicar los tiempos de ejecución obtenidos, los cuales se detallan en la tabla 4.3. Para 10^4 elementos, la eficiencia

Versión	10^3	10^4	10^5
Haskell no mutable	20 ms	600 ms	—
Haskell mutable	20 ms	240 ms	3580 ms
Clean	3 ms	32 ms	360 ms
C	2 ms	20 ms	190 ms

Tabla 4.3: Tiempos de ejecución de la función *insert*

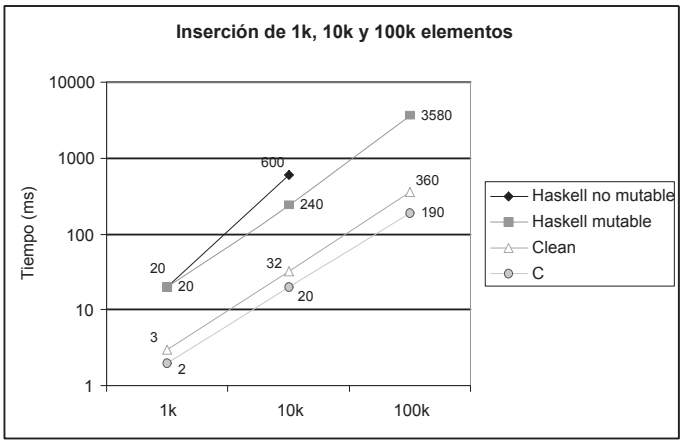

Figura 4.9: Gráfico de los tiempos de ejecución de la Tabla 4.3

de la versión Haskell mutable es muy superior a la de la versión no mutable, introduciendo una mejora del 60%. Sin embargo, la implementación basada en arreglos mutables es entre 10 y 18.84 veces más lenta que la versión C.

La versión Clean es la que presenta el menor tiempo de ejecución de todas las versiones funcionales, dado que dicho lenguaje provee la implementación de arreglos mutable más eficiente. Los tiempos de ejecución obtenidos por este lenguaje rivalizan con los obtenidos por la implementación C. La diferencia entre ambas versiones para la inserción de 10^3 elementos es de sólo 1 ms. Para la entrada de tamaño 10^4, la mejora introducida por la versión C es sólo del 37.5%, y 47.22% para 10^5 elementos. Este resultado es muy satisfactorio, sobre todo considerando que la eficiencia temporal de los lenguajes funcionales puros tiende a disminuir rápidamente para entradas grandes.

La Fig. 4.9 muestra un gráfico comparativo de los tiempos obtenidos. Nuevamente se utilizó una escala logarítmica para el eje y, de modo que el gráfico refleje la verdadera magnitud de las curvas. Notar cómo la curva correspondiente a la versión Clean prácticamente se superpone con la que da los tiempos de ejecución de la implementación C.

En la Secc. 4.1 se muestra que el tiempo de ejecución para una única

$f_T(i)$	$i = 3$	$i = 4$	$i = 5$
Haskell no mutable	0.632	0.600	∞
Haskell mutable	6.029	6.430	8.830
Clean	0.904	0.857	0.888
C	0.602	0.535	0.468

Tabla 4.4: Constantes de proporcionalidad de complejidad temporal

inserción sobre un VEBTree de orden u es $O(\log\log(u))$. Este resultado vale para las versiones Haskell mutable, Clean y C, pero no para la versión Haskell no mutable. Debido a que el arreglo de *widgets* se actualiza en tiempo $O(\sqrt{u})$, la inserción completa tiene también dicha complejidad. Por lo tanto, u inserciones demandan $O(u \log\log(u))$ pasos para todas las versiones salvo la implementación Haskell no mutable, que toma $O(u\sqrt{u})$ pasos.

Para analizar la relación entre estas fórmulas de complejidad y los tiempos listados en la Tabla 4.3, se llevará a cabo el mismo procedimiento que en la Secc. 4.5.1. Esto es, siendo $T(i)$ el tiempo de ejecución que toma insertar $u = 10^i$ elementos en un VEBTree de orden u, se analizarán los valores resultantes de la aplicación de la función $f_T(i) = 10^3 \frac{T(i)}{T_c(i)}$, $i \in \{3, 4, 5\}$, donde $T_c(i) = 10^i\sqrt{10^i}$ para la versión Haskell no mutable, y $10^i \log\log(i)$ para las restantes. El producto por 10^3 es para evitar que f_T tome valores demasiado pequeños, evitando la pérdida de precisión por el sistema de numeración de punto flotante. La Tabla 4.4 lista los resultados obtenidos.

Las versiones Haskell no mutable y Clean presentan un comportamiento casi constante, de acuerdo a lo esperado. La versión Haskell mutable presenta un crecimiento considerable para $i = 10^5$, que puede justificarse por la escasa eficiencia temporal y espacial de los arreglos mutables implementados por dicho lenguaje. En efecto, la Fig. 4.6 muestra que la función que consume más memoria es *ins*, la cual es la única que opera sobre el arreglo de *widgets*. Además, la constante de proporcionalidad para esta versión es mucho mayor que las restantes, lo que también puede atribuirse al costo de las operaciones sobre este tipo de arreglos. Las constantes de proporcionalidad obtenidas por la versión C también muestran un comportamiento interesante: conforme crece el tamaño de la entrada, muestran una tendencia levemente decreciente. Esto se explica porque los VEBTrees están implementados con arreglos, que C modela en el heap mediante áreas de memoria continuas. De esta manera, los VEBTrees imperativos gozan de la propiedad de *localidad*, la cual establece que las direcciones de memoria con alta probabilidad de ser accedidas a corto plazo son las cercanas a las más recientemente direccionadas. Esto hace que la versión C se vea beneficiada por las optimizaciones por cache que provee

el manejo de memoria de Linux (el *slab allocator schema* [BC01]), que basan su efectividad en esta propiedad.

4.5.3 Conclusiones

Los resultados obtenidos muestran que es posible implementar estructuras de datos procedurales bajo el paradigma funcional puro con resultados comparables a los obtenidos por un lenguaje procedural, para conjuntos de datos razonablemente grandes (del orden de 10^5 elementos). También permiten demostrar la viabilidad de utilizar un enfoque *ecléctico* para desarrollar estructuras de datos: los lenguajes funcionales puros permiten añadir de manera segura características imperativas al paradigma, de este modo es posible obtener estructuras funcionales puras con un alto grado de eficiencia.

En general, la estructuras de datos imperativas actualizan *in place* un buffer. La única manera de emular un buffer con estas características bajo un entorno funcional puro es mediante arreglos mutables, por lo que esta abstracción es el componente principal de dichas estructuras. Por lo tanto, los arreglos mutables eficientes son fundamentales para obtener tiempos de ejecución comparables a los conseguidos por los lenguajes imperativos. Esto es particularmente cierto para los VEBTrees, cuya eficiencia depende enteramente de los arreglos mutables utilizados en su implementación.

Los resultados conseguidos demuestran que los arreglos provistos por Clean son *mucho* más eficientes que los implementados por Haskell. Por este motivo, la versión Clean es la más eficiente de las tres implementaciones funcionales puras presentadas en este capítulo. En el peor caso (para la entrada de 10^5 elementos) sólo ejecuta 1.89 veces más lento que la versión C. Este alto grado de eficiencia se explica también por las anotaciones de *strictness*, que permiten controlar de manera indirecta el consumo de memoria de un programa. Sin embargo, la versión Clean es mucho menos clara que la versión Haskell. Esto se debe a que es responsabilidad del programador mantener la unicidad de los arreglos, lo que tiende a complicar el código con las funciones provistas por Clean para operar sobre arreglos *unique*. Por el contrario, las anotaciones de *strictness* no tienen un impacto significativo sobre la claridad de la implementación Clean.

La eficiencia espacial es un factor que debe ser tenido en cuenta si se utiliza un lenguaje con manejo automático de memoria, como los lenguajes funcionales puros. Un mal uso de la memoria provoca la ejecución de las rutinas de *garbage collection*, lo que hace que un programa desperdicie tiempo de ejecución liberando el heap. La comparación entre el rendimiento espacial de las versiones Haskell no mutable (*lazy*) y mutable (estricta), dadas por las Figs. 4.6 y 4.7, muestran que la diferencia de consumo de memoria entre una

implementación *lazy* y otra parcialmente estricta de una misma estructura de datos puede ser importante. Para este ejemplo en particular, la implementación mutable necesita aproximadamente un 50% menos de la memoria requerida por la no mutable. Esto permite concluir que para implementar estructuras de datos procedurales eficientes, un lenguaje funcional puro debe proveer buenos mecanismos para denotar porciones de código como estrictas.

Debe considerarse que en el lenguaje C la memoria se maneja explícitamente, i.e., no hay *garbage collection*. O sea, los VEBTrees C no liberan memoria durante su ejecución a no ser que tal operación sea forzada desde el código. Como los casos de prueba consisten en efectuar una cantidad fija de inserciones, jamás se pierde tiempo liberando memoria, al contrario de lo que sucede con las versiones Haskell y Clean. Esto es un factor a favor la versión C que debe ser tenido en cuenta al momento de analizar los resultados obtenidos.

Capítulo 5

Conclusiones y Trabajo Futuro

A lo largo de este trabajo se estudiaron diversas técnicas para la implementación de estructuras de datos no triviales bajo un entorno funcional puro. El estudio se dividió en dos partes. En la primera se discutieron los aspectos de implementación de dos estructuras funcionales puras: árboles AVL y Treaps. En la segunda se estudiaron los árboles de van Emde Boas, estructura imperativa que permite modelar con eficiencia subconjuntos de universos finitos cuyos elementos soportan aritmética entera. Para asegurar diversidad, se utilizaron dos lenguajes funcionales diferentes: Haskell y Clean. Las diferencias existentes entre ambos impactan de manera considerable en las técnicas de implementación utilizadas, sobre todo para los árboles de van Emde Boas. El estudio se completó mediante una fase de análisis de eficiencia. Mediante herramientas de *profiling*, se analizó la eficiencia tanto temporal como espacial de las estructuras de datos funcionales implementadas, comparando los resultados obtenidos entre sí. Uno de los objetivos de esta tesis es obtener estructuras de datos funcionales eficientes, con tiempos de ejecución comparables a los obtenidos por una implementación imperativa; por lo tanto, para cada estructura de datos se implementó una versión C, de modo que fuera posible efectuar dichas comparaciones.

Los resultados obtenidos son satisfactorios. Exceptuando la versión Haskell mutable de VEBTrees, el peor tiempo de ejecución de una implementación funcional en relación al obtenido por el lenguaje C es sólo 5.56 veces más lento. Esta pequeña pérdida de performance se ve compensada por diversas ventajas que otorga el paradigma funcional (por ejemplo, código más claro y fácil de mantener) que serán analizadas a lo largo de este capítulo.

A priori, se destacan dos conclusiones principales. Por un lado, este trabajo muestra que es posible desarrollar estructuras de datos bajo un entorno funcional puro con eficiencia comparable a la de los lenguajes imperativos, para entradas de datos razonablemente grandes. Este resultado es importante,

pues estructuras de datos eficientes permiten codificar algoritmos eficientes. A su vez, el capítulo 4 prueba que es factible adoptar un enfoque ecléctico para implementar eficientemente estructuras de datos procedurales (o sea, mediante la combinación de construcciones funcionales puras con efectos imperativos tales como el *update in place*).

Los lenguajes funcionales puros proveen además dos ventajas adicionales: la posibilidad de demostrar correctitud vía razonamiento ecuacional (tal como se hizo para árboles AVL y Treaps), y un sistema de tipos avanzado que permite expresar polimorfismo de manera natural, sin necesidad de recurrir a características oscuras de los lenguajes imperativos, tales como los punteros a función.

Las actividades de implementación, *debugging* y *profiling* desarrolladas a lo largo de esta tesis aportan conclusiones que pueden ser clasificadas según diversos aspectos: diferencias entre los lenguajes utilizados, importancia del análisis de eficiencia, ventajas y desventajas del uso de *lazy evaluation*, beneficios aportados por el sistema de tipos, y propiedades de derivación y razonamiento ecuacional asociadas a los lenguajes funcionales puros.

Haskell vs. Clean

El uso intensivo de Haskell y Clean muestra que los ejecutables Clean son mucho más eficientes que los obtenidos mediante Haskell. Esta conclusión se aplica tanto a la eficiencia temporal como espacial. Los motivos principales que explican esta diferencia son lo siguientes:

- **Implementación:** la implementación de Clean, dada por un sistema de reducción de grafos conocido como la *ABC machine* [BS92], es más eficiente que el sistema implementado por el compilador Haskell (la *G–machine* [PJ87]).

- **Anotaciones de *Strictness*:** los análisis de eficiencia efectuados a lo largo de esta tesis muestran que la capacidad de especificar de manera flexible los fragmentos de código que deben evaluarse de modo estricto tiene un impacto directo en la eficiencia de una implementación. Clean provee un mecanismo para especificar *strictness* mucho más flexible que el provisto por Haskell.

- **Arreglos Mutables:** el análisis de eficiencia del capítulo 4 demuestra que los arreglos mutables provistos por Clean son más eficientes que los implementados por Haskell. En Clean, la eficiencia de los arreglos mutables eficientes es una prioridad en el diseño del lenguaje. En Haskell, son una construcción más, implementable mediante mónadas.

No obstante, dado que los *unique types* no proveen el encapsulamiento provisto por las mónadas, en Clean el estado que modela a un arreglo mutable debe ser manipulado explícitamente. La invariancia de la unicidad es entonces responsabilidad del programador, por lo que los arreglos mutables Clean son más difíciles de utilizar que los provistos por Haskell.

Herramientas de *Profiling* y Consumo de Memoria

Las herramientas de *profiling* son un mecanismo fundamental para obtener programas funcionales eficientes, ya que permiten detectar qué fragmentos de código causan comportamiento ineficiente. Si bien esta afirmación es aplicable a cualquier lenguaje de programación sin importar el paradigma, las técnicas de *profiling* utilizadas para los lenguajes funcionales puros difieren de las aplicadas otros lenguajes.

Bajo el paradigma funcional puro, las herramientas de *heap profiling* cobran una importancia mucho mayor a la que poseen bajo otros paradigmas, donde los resultados dependen en mayor medida de la eficiencia temporal. Si bien los compiladores funcionales producen código cuya eficiencia temporal es comparable a la de los lenguajes imperativos, no sucede lo mismo con la eficiencia espacial. Los requerimientos de memoria por parte de los lenguajes funcionales tienden a ser elevados. Un programa que utilice más memoria de la disponible ejecutará repetidamente fases de administración de memoria (*garbage collection*), lo que deteriorará su eficiencia temporal. Las pruebas de rendimiento efectuadas a lo largo de este trabajo muestran que en todos los casos donde la eficiencia no es la esperada, el motivo es la ejecución de fases de *garbage collection*. Por lo tanto, es vital mantener bajo control el consumo de heap. Aunque un mal uso de memoria puede mejorarse en general mediante anotaciones de *strictness*, es importante determinar con precisión que porciones de código deben ser modificadas. En consecuencia, un *heap profiler* es una herramienta esencial para obtener código funcional espacial y temporalmente eficiente [ML98].

Lazy Evaluation

La *lazy evaluation* suele ser una de las causas principales que justifican un consumo de memoria elevado. Esto es porque los lenguajes *lazy* almacenan en el heap las computaciones cuya ejecución ha sido diferida (*closures*). Sin embargo, no debe concluirse que la *lazy evaluation* es una característica no deseable de los lenguajes funcionales puros. Por el contrario, permite expresar soluciones a problemas complejos de manera concisa y elegante; por

ejemplo, en la Secc. 3.2 se la utiliza para implementar un *stream* infinito de números pseudo–aleatorios, efectuando una operación esencialmente imperativa sin utilizar mónadas o *unique types*. La literatura muestra numerosos casos donde su uso resulta provechoso. Por ejemplo, mediante técnicas de *amortización* basadas en *lazy evaluation*, Okasaki implementa estructuras de datos altamente eficientes tales como *deques* y *catenable lazy lists* [Oka96b]. Además, la *lazy evaluation* disminuye el grado de cohesión de un programa, dado que las funciones puede programarse independientemente de otras, sin necesidad de presuponer un orden de evaluación particular [Hug89].

En consecuencia, la *lazy evaluation* es una propiedad deseable de los lenguajes funcionales puros que debe ser explotada. Sin embargo, es importante evitar usos innecesarios y restringirla al dominio adecuado. Por otra parte, los lenguajes funcionales proveen los mejor de ambos mundos. La *lazy evaluation* es una característica muy difícil de emular mediante un lenguaje imperativo. En cambio, suele ser sencillo forzar el orden de evaluación estricto bajo el paradigma funcional puro.

Polimorfismo Paramétrico y Ad-hoc

El sistema de tipos provisto por los lenguajes funcionales modernos representa una gran ventaja con respecto a los lenguajes imperativos. Esta tesis muestra que los lenguajes funcionales puros permiten modelar fácilmente el carácter polimórfico de las estructuras de datos implementadas, mientras que en C dicho efecto se logra mediante punteros `void` para los elementos y a función para los comparadores. Esto es una desventaja evidente si se desea almacenar tipos primitivos, dado que se fuerza a trabajar con punteros a dichos tipos en lugar de los valores propiamente dichos.

Si bien lenguajes como Ada [Gue97] o C++ [Str94] permiten expresar polimorfismo paramétrico mediante el uso de *generic packages* y *templates* respectivamente, aún están en desventaja con respecto a Haskell y Clean debido a que no pueden expresar adecuadamente polimorfismo ad-hoc [CW85]. Haskell y Clean permiten expresarlo mediante su sistema de clases [Jon95]. Una clase define un conjunto de operaciones, y todo tipo que sea instancia de dicha clase debe proveer una implementación para las funciones que la componen.

El polimorfismo ad-hoc puede ser emulado en Java mediante interfaces (con limitaciones debido a la existencia de tipos primitivos y la ausencia de *operator overloading*), y por una extensión no standard de C++ llamada *signatures* [BR95]. No obstante, estos mecanismos no logran alcanzar el poder expresivo del sistema de tipo de los lenguajes funcionales puros. En conclusión, el sistema de tipos altamente expresivo de los lenguajes funcionales

puros se manifiesta como una ventaja sumamente importante frente a los lenguajes imperativos y orientados a objetos utilizados comúnmente en la industria.

Correctitud

Debido a la ausencia de efectos laterales, los lenguajes funcionales puros permiten razonar ecuacionalmente sobre los programas. Esto es una ventaja muy importante sobre los lenguajes procedurales, donde la correctitud debe probarse mediante un sistema formal consistente con la semántica de dichos lenguajes. Si bien esta afirmación no es válida para los programas monádicos (dado que las mónadas introducen efectos laterales, aunque sin violar la transparencia referencial), existen técnicas de razonamiento sobre mónadas, aunque menos intuitivas [BMV99]. Por otra parte, es posible limitar el uso de mónadas a las porciones de código donde son realmente necesarias, por lo que las técnicas de razonamiento ecuacional siguen siendo aplicables al código restante.

El razonamiento ecuacional permite utilizar técnicas algebraicas sencillas tales como la inducción estructural para probar resultados de correctitud. Por el contrario, el razonamiento sobre lenguajes imperativos es engorroso, implica la búsqueda de invariantes adecuados (lo cual no suele ser sencillo), y no es una práctica común en la industria. Otro beneficio proveniente del razonamiento ecuacional lo constituyen las técnicas de *derivación*. Gracias a la ausencia de efectos laterales, los lenguajes funcionales puros permiten obtener sucesivamente programas equivalentes más eficientes a partir de una formulación inicial simple y correcta, pero ineficiente.

Esta tesis pone en práctica las técnicas de razonamiento ecuacional en los capítulos 2 y 3, donde las propiedades que definen a los árboles AVL y a los treaps bien formados se expresan mediante predicados de primer orden, y luego se prueba que las implementaciones dadas preservan dichas fórmulas.

5.1 Trabajo Futuro

Aunque la eficiencia de las estructuras de datos funcionales estudiadas es satisfactoria comparada con los tiempos de ejecución obtenidos mediante el lenguaje C, existen aún fragmentos de código ineficientes que deben ser mejorados. La eficiencia de las versiones Haskell es aceptable, pero baja comparada con los resultados obtenidos mediante Clean. Por lo tanto, debe profundizarse particularmente el estudio del rendimiento de las implementaciones Haskell. Existen dos técnicas de optimización dependientes del compilador

Haskell que no han sido utilizadas, y que podrían aportar alguna mejora. Se tratan de los *pragmas* de compilación y las directivas de *inlining*. Los *pragmas* son directivas de compilación para generar código más eficiente; el *inlining* consiste en reemplazar una función por su código, evitando así una llamada a función.

El `nhc` [Röj94] es un compilador para un subconjunto de Haskell, desarrollado para permitir la implementación de programas espacialmente eficientes. Además, provee herramientas para *heap profiling* mucho más sofisticadas que las provistas por el compilador Haskell utilizado en esta tesis (`ghc` [PJHH+93]). Si bien el código producido por este compilador es más lento que el generado por el `ghc`, su uso sería de utilidad para detectar fragmentos de código que produzcan un comportamiento espacial ineficiente, que no hayan sido detectadas por los *heap profilers* standard.

Aún es una práctica común dentro de la comunidad funcional desarrollar estructuras de datos ad-hoc, o hacer uso de abstracciones como listas, simples pero no necesariamente eficientes [PJ97]. Si bien ya existen algunas bibliotecas funcionales puras de estructuras de datos (por ejemplo, Edison [Oka99]), estas tienden a ser incompletas, ya que no implementan estructuras de datos imperativas utilizadas comúnmente, tales como *hash tables*. Es importante definir las bases para una biblioteca de estructuras de datos funcional pura standard (tal como STL para C++), que incluya las estructuras de datos implementadas en este trabajo, y que ponga énfasis en las estructuras de datos procedurales.

Existen otros lenguajes contra los que sería provechoso comparar las implementaciones funcionales desarrolladas a lo largo de esta tesis. Dos de ellos son Java [AGH01] y Smalltalk [GR83]. Estos lenguajes son aptos para efectuar comparaciones de rendimiento por dos motivos. Al igual que los lenguajes funcionales, ambos proveen manejo automático de memoria, por lo que la comparación sería más exacta que contra C. Por otra parte, estos dos lenguajes (especialmente Java) forman parte del *mainstream* actual, siendo ampliamente utilizados para proyectos de software comerciales a gran escala. Pueden esperarse buenos resultados, ya que ambos lenguajes son más lentos que C. Por ejemplo, la implementación más avanzada de la JVM (la *Hotspot* JVM [Inc01]) funciona en el mejor de los casos 1.5 veces más lento que C++, lo cual llega a ser superado por las tres implementaciones Clean estudiadas.

Apéndice A

Introducción a los Lenguajes Funcionales

Este apéndice presenta una breve introducción a los conceptos básicos del paradigma funcional puro, enfatizando los utilizados a lo largo de este trabajo. Dicha introducción está basada principalmente en el lenguaje Haskell, debido a que actualmente es el lenguaje funcional standard. Clean está fuertemente inspirado en los principios de diseño de Haskell, por lo que los conceptos discutidos se aplican también a dicho lenguaje con pocas o sin modificaciones. Sin embargo, Clean presenta diferencias importantes que, por motivos de claridad, son discutidas separadamente

El objetivo principal de este apéndice es proveer una base mínima de conocimiento necesario para comprender completamente las implementaciones dadas a lo largo de este trabajo. Por lo tanto, hay numerosos temas fundamentales que se dejarán de lado. El lector con un interés que supere las ideas discutidas aquí, puede remitirse a [Dav92, Tho96, Bir98], o cualquier otro libro de introducción a los lenguajes funcionales.

A.1 Funciones

El paradigma funcional se basa en la computación de expresiones, entidades básicas que permiten definir a las funciones que componen un programa. El valor de una expresión depende únicamente de las subexpresiones que la conforman, esto es, no existen efectos laterales que puedan afectar dicho valor. Esta propiedad es una característica fundamental del paradigma funcional puro, denominada *transparencia referencial*. La importancia de esta propiedad radica en que garantiza que siempre que una subexpresión sea reemplazada por otra que retorne el mismo valor, el valor retornado por la

expresión principal no se verá afectado. Esto hace posible razonar ecuacionalmente sobre un programa funcional, sin que el proceso se vea afectado por detalles de implementación.

Dado que un programa se compone de funciones, la aplicación será una operación efectuada a menudo. Por este motivo, en los lenguajes funcionales la aplicación se denota mediante yuxtaposición, esto es, $f\ x$ representa a una expresión cuyo resultado es aplicar la función f a un parámetro x. Si el parámetro no es una expresión atómica, se deben utilizar paréntesis para evitar que la aplicación se realice sobre una parte de la expresión denotada por x. Por ejemplo, asumiendo que f recibe números enteros, la aplicación de f al sucesor de x se escribe como $f\ (x+1)$ y no $f\ x+1$, que denota sumar 1 al resultado de aplicar f a x. Sea sq la función que recibe un número entero y retorna dicho número elevado al cuadrado. Una manera de definirla es:

$$sq :: Int \rightarrow Int$$
$$sq\ x = x * x$$

Notar la declaración de tipo que antecede a la definición de sq. Dicha declaración formula que sq es una función que recibe un parámetro de tipo entero, y retorna un valor del mismo tipo.

Los lenguajes funcionales son sumamente expresivos, y es común que existan diversas maneras de escribir una misma expresión. Las implementaciones estudiadas en esta tesis se basan mayormente en cuatro construcciones: expresiones *let* e *if*, cláusulas *where*, y definición por alternativas disjuntas. Existe una quinta alternativa, las definiciones por *pattern matching*, que serán discutidas en la Secc. A.2.

Las expresiones *let* se componen de declaraciones auxiliares y un cuerpo. Las declaraciones auxiliares tienen su alcance limitado al cuerpo del *let*. Por ejemplo, la función que eleva a un elemento a la cuarta se puede implementar usando un *let* de la siguiente manera:

$$fourth :: Int \rightarrow Int$$
$$fourth\ x = \textbf{let}\ x_1 = sq\ x\ \textbf{in}\ sq\ x_1$$

La declaración auxiliar define a x_1 como el cuadrado de x. Dicha expresión se utiliza en el cuerpo del *let* para computar la cuarta potencia de x. La cláusula *where* es otra construcción que permite declarar funciones auxiliares. Al igual que con los *let*'s, su alcance se limita al cuerpo principal de la función que define el *where*. Usando estas cláusulas, *fourth* se podría escribir así:

$$
\begin{aligned}
&fourth :: Int \rightarrow Int \\
&fourth\ x = x_1 * x_1 \\
&\quad \textbf{where}\ x_1 = sq\ x
\end{aligned}
$$

La diferencia más importante entre los *let*'s y *where* es que *let* es una expresión, y *where* no. Una función puede definirse usando únicamente un *let*, pero no sólo con un *where*: una definición de la forma $fx = where \ldots$ es inválida.

La expresión *if* permite definir expresiones que serán evaluadas condicionalmente, en función del valor retornado por una expresión booleana. Si bien en apariencia un *if* es similar a su homónimo imperativo, existe una diferencia importante: el *if* funcional es una expresión, mientras que el imperativo es un comando. Como ejemplo se implementará la función *sign*, que recibe un entero y retorna 1, 0, ó -1 según sea mayor, igual o menor que 0 respectivamente.

$$
\begin{aligned}
&sign :: Int \rightarrow Int \\
&sign\ x = \textbf{if}\ x == 0\ \textbf{then}\ 0\ \textbf{else}\ div(x, abs\ x)
\end{aligned}
$$

La función *div* retorna el resultado de la división entera entre sus dos argumentos, y *abs*, el valor absoluto del número recibido. Otra opción para definir a *sign* consiste en utilizar alternativas disjuntas. En este caso se definen un conjunto de alternativas para la función, cada una asociada con una expresión booleana. Al ser invocada, la función evalúa dichas expresiones, eligiendo la alternativa asociada a la expresión cuya evaluación retorne *True*.

$$
\begin{aligned}
&sign :: Int \rightarrow Int \\
&sign\ x \\
&\quad \mid x == 0 \quad\ = 0 \\
&\quad \mid \textbf{otherwise} = div(x, abs\ x)
\end{aligned}
$$

La palabra clave *otherwise* equivale a *True*, y permite seleccionar dicha alternativa cuando no se satisface ninguna de las anteriores.

A.1.1 Funciones como Valores

Una característica importante de los lenguajes funcionales es el uso de funciones como valores del lenguaje: las funciones pueden pasarse como parámetro, retornarse como resultado, o almacenarse en estructuras de datos (incluso *ser* estructuras de datos). Esta propiedad permite definir funciones altamente reusables. Por ejemplo, considérese la función *newton*, que retorna una aproximación a la derivada de una función cualquiera en un punto dado mediante el cociente incremental de Newton:

```
newton :: (Float -> Float, Float) -> Float
newton (f, p) = (f (p + epsilon) - f p) / epsilon
  where
    epsilon = 0.00001
```

La propiedad más importante de esta función es que permite calcular la derivada de *cualquier* función sin necesidad de modificar el código en modo alguno. Notar la definición de tipo, donde el parámetro correspondiente a la función a derivar se denota mediante una declaración de función ordinaria, y el uso de un par ordenado para modelar a los dos parámetros recibidos.

A.1.2 Currificación

Las funciones de alto orden permiten razonar de un modo alternativo sobre las funciones. Sea *sum* una función que computa la suma de dos números enteros:

```
sum :: (Int, Int) -> Int
sum (x, y) = x + y
```

En lugar de esta definición, puede darse una nueva donde se toma un único parámetro de tipo entero y se devuelve una función que espera otro entero y retorna el resultado de computar la suma de ambos.

$$sum :: Int \rightarrow (Int \rightarrow Int)$$
$$(sum\ x)\ y = x + y$$

La declaración de tipo consigna que *sum* toma un número entero y retorna una función del tipo $(Int \rightarrow Int)$. De este modo, la expresión $(sum\ x)$ es una función que recibe un número entero y, y retorna $x + y$. Debido a que la aplicación de funciones es asociativa a izquierda y a que las signaturas de tipo lo son a derecha, tanto los paréntesis de la definición de *sum* como los de su tipo pueden eliminarse. Luego, la función puede definirse $sum\ x\ y = x + y$, y la signatura se puede escribir como $Int \rightarrow Int \rightarrow Int$. La transformación de la primera versión de *sum* en la segunda se denomina *currificación*. La ventaja de la currificación consiste en que una función puede tomar sus parámetros de uno a la vez, y la expresión resultante sigue siendo válida. Esto permite formas muy elegantes de reutilización de código. Por ejemplo, la versión currificada de *newton* permite definir la función *newtonTwice*, que calcula la derivada segunda de una función dada.

$$newtonTwice :: (Float \rightarrow Float) \rightarrow Float \rightarrow Float$$
$$newtonTwice\ f\ p = newton\ (newton\ f)\ p$$

Puede verse que sería mucho más complicado obtener una definición de *newtonTwice* basada en la versión original de *newton*. Notar que el parámetro p aparece al final en ambos lados de la ecuación. Dado que los lenguajes funcionales permiten el razonamiento ecuacional, p puede "cancelarse" de ambos lados de la igualdad sin afectar la validez de la definición, que pasa a ser: $newtonTwice\ f = newton\ (newton\ f)$. A esta cancelación se la conoce como η-reducción, y se dice que las dos definiciones de *newtonTwice* son η-equivalentes.

A.1.3 *Lazy Evaluation*

La *lazy evaluation* es un mecanismo de evaluación que sigue la regla de "computar sólo si es necesario, y en tal caso, sólo una vez". La *lazy evaluation* disminuye el grado de cohesión entre funciones y aumenta la reusabilidad

del código, dado que las funciones pueden desarrollarse independientemente unas de otras si necesidad de tener en cuenta el orden de evaluación [Hug89]. Este orden de evaluación es propiedad exclusiva de los lenguajes funcionales puros, siendo muy difícil de emular en mediante un lenguaje imperativo, pues no es compatible con la presencia de efectos laterales. Como ejemplo de utilización de la *lazy evaluation*, considérense las siguientes definiciones:

$$
\begin{aligned}
&fst :: (Float, Float) \rightarrow Float \\
&fst\ (x, y) = x \\
&noError = fst\ (1, 1/0)
\end{aligned}
$$

Bajo un orden de evaluación estricto *noError* fallaría, ya que la evaluación de 1/0 no está definida. Pero como la segunda componente del par no es referenciada, no hace falta computarla y la función *noError* retorna 1.

Otra ventaja de la *lazy evaluation* es el uso de estructuras infinitas, por ejemplo, listas. Dado un tipo a, $[a]$ representa una lista con elementos de dicho tipo. Una lista puede estar vacía (representada mediante la expresión [ ]), o bien puede estar formada por un elemento x seguido de una lista (posiblemente vacía) xs; esta construcción se escribe como $(x : xs)$. Sea *ones* una lista infinita de unos:

$$
\begin{aligned}
&ones :: Int \rightarrow [Int] \\
&ones = 1 : ones \quad \text{— } ones = [1,1,\ldots]
\end{aligned}
$$

La función *length* calcula la longitud de una lista dada. Como esta operación requiere recorrer la lista en su totalidad, *length* es estricta en su argumento. Por lo tanto, (*length ones*) referencia a *ones* en un contexto estricto, y su resultado no está definido. Por otra parte, la función *take* recibe un entero n y una lista, y retorna los primeros n elementos de dicha lista. Esta función no es estricta sobre la lista, ya que sólo necesita obtener los primeros n elementos. Por lo tanto, (*take n ones*) utiliza a *ones* en un contexto *lazy*, y la computación puede efectuarse. Observar que (*take n ones*) no estaría definida en un lenguaje estricto.

A.2 Sistema de Tipos

En general, los lenguajes funcionales puros son fuertemente tipados. Tanto Haskell como Clean implementan sistemas de tipos estáticos, lo que significa que se asigna un tipo a toda expresión en tiempo de compilación. Además, ambos lenguajes proveen *inferencia de tipos*. Mediante este mecanismo el compilador puede intentar determinar el tipo de toda función para la que no se provean las signaturas correspondientes, o informar del error correspondiente si la signatura dada no coincide con la inferida. Algunas de las características principales de los sistemas de tipos funcionales son el polimorfismo paramétrico, los tipos algebraicos, y las clases de tipos para polimorfismo ad-hoc (*overloading*).

A.2.1 Polimorfismo paramétrico

El polimorfismo paramétrico es la capacidad de una función para operar sin conocer el tipo de sus parámetros. Esto permite definir funciones que operen sobre conjunto infinitos de tipos, permitiendo de este modo un alto nivel de reusabilidad. Un ejemplo clásico lo constituye *id*, la función identidad.

$$id :: a \rightarrow a$$
$$id\ x = x$$

Notar que la función *id* no precisa conocer cuál es el tipo de la variable a, que puede ser instanciada a cualquier tipo válido. Por lo tanto, la definición de *id* puede ser vista en realidad como la definición de *infinitas* funciones identidad, una para cada tipo.

El polimorfismo paramétrico provee ventajas tales como reusabilidad y generalidad, ya que permite definir funciones que operan sobre infinitos tipos. También favorece la programación modular, ya que permite definir módulos de utilidad general, sin necesidad de tener en cuenta las aplicaciones donde serán utilizados.

A su vez, el polimorfismo paramétrico puede usarse para modelar funciones como valores del lenguaje. De este modo es posible dar definiciones sumamente generales tales como la composición de funciones, implementada por la función *compose*, o la función *map*, que toma una función y una lista, y aplica la función a cada elemento de la lista produciendo una nueva.

```
compose :: (b → c) → (a → b) → a → c
compose f g x = f (g x)

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

Estas dos funciones son usadas virtualmente en todo programa funcional, y existen muchas funciones más que explotan el polimorfismo paramétrico para proveer generalidad. En caso que se desee estudiar estos ejemplos, se recomienda recurrir a la bibliografía citada al comienzo del apéndice.

Este trabajo hace uso extensivo del polimorfismo paramétrico para definir las estructuras de datos estudiadas de manera genérica, estando todas las implementaciones funcionales parametrizadas en el tipo de los elementos a ser almacenados.

A.2.2 Tipos Algebraicos

Los lenguajes funcionales permiten la creación de nuevos tipos mediante los *tipos algebraicos*. El programador crea un tipo algebraico especificando su nombre y las maneras de construir valores del tipo en cuestión. Por ejemplo, el tipo *Bool* puede definirse de la siguiente manera:

```
data Bool = False | True
```

Este mecanismo puede utilizarse para expresar definiciones de tipo mediante inducción. Un ejemplo típico lo constituyen los número naturales. Inductivamente, un número natural puede ser 0, o el sucesor de un número natural. Esto resulta en la siguiente definición:

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

Esto es, el conjunto de números naturales se define mediante dos constructores, uno para su caso base y otro para el caso recursivo. Los tipos algebraicos admiten también polimorfismo paramétrico. De esto modo, es posible definir "estructuras" parametrizables con un tipo cualquiera tales como el tipo *Maybe a*, utilizado a lo largo de este trabajo para modelar computaciones que pueden fallar de manera controlada. Mediante este tipo es posible definir una función de división que falle adecuadamente si el divisor es cero:

$$data\ Maybe\ a\ =\ Nothing\ \mid\ Just\ a$$

$$safeDiv\ ::\ Float\ \rightarrow\ Float\ \rightarrow\ Maybe\ Float$$
$$safeDiv\ x\ y\ =\ \textbf{if}\ y\ ==\ 0\ \textbf{then}\ Nothing\ \textbf{else}\ Just\ (x/y)$$

Los árboles constituyen otra estructura que puede ser definida elegantemente mediante tipos algebraicos. Por ejemplo, un árbol binario puede ser un árbol vacío o bien un nodo cuyos hijos son dos árboles binarios.

$$\textbf{data}\ BinTree\ a\ =\ Empty\ \mid\ Node\ a\ (BinTree\ a)\ (BinTree\ a)$$

Dado que los tipos algebraicos están constituidos por diferentes alternativas de construcción, los lenguajes funcionales deben proveer un mecanismo que permita a una función distinguir entre ellas. Este mecanismo se conoce como *pattern matching*, y hace posible asociar una expresión a computar para cada alternativa de las expresiones recibidas como parámetro. Por ejemplo, la función que calcula la altura de un árbol binario dado por el tipo (*Bintree a*) puede definirse mediante *pattern matching* así:

$$height\ ::\ Bintree\ a\ \rightarrow\ Int$$
$$height\ Empty\ =\ 0$$
$$height\ (Node\ \_\ t_1\ t_2)\ =\ 1\ +\ max\ (height\ t_1)\ (height\ t_2)$$

El mecanismo de *pattern matching* implementado por el compilador intentará hacer coincidir el argumento recibido contra ambas alternativas, aplicando el código asociado a la alternativa para la cual la comparación resulte exitosa. De este modo, para un árbol vacío *height* retornará 0, mientras que para un árbol no vacío retornará 1 más la altura de subárbol más profundo. Como *height* calcula la altura de un árbol dado, no interesan los valores almacenados en los nodos. Para denotar este hecho se utiliza el símbolo "_", el cual representa a un parámetro que no será referenciado en el lado derecho de la definición.

Otra manera de definir funciones basadas en *pattern matching* es mediante expresiones *case*. Esta construcción resulta de utilidad para definir las distintas alternativas de una función mediante una única expresión. Por ejemplo, utilizando un *case* es posible definir a *height* con una sola expresión, en lugar de las dos utilizadas en la definición anterior:

$$
\begin{array}{l}
height :: Bintree\ a \rightarrow Int \\
height\ t = \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad Empty \rightarrow 0 \\
\qquad\qquad Node\ \_\ t_1\ t_2 \rightarrow 1 + max\ (height\ t_1)\ (height\ t_2)
\end{array}
$$

A veces puede ser necesario referenciar simultáneamente un parámetro y parte de él. En estos casos resulta conveniente utilizar un @–*pattern*, el cual permite definir un sinónimo para un argumento dado, por ejemplo, t es sinónimo del árbol $(Node\ x\ t_1\ t_2)$ en $t@(Node\ x\ t_1\ t_2)$. Sea *replace* una función que dado un predicado, un valor n y un árbol binario, reemplaza su raíz por n si el predicado se cumple para dicho valor.

$$
\begin{array}{l}
replace :: (a \rightarrow Bool) \rightarrow a \rightarrow BinTree\ a \rightarrow BinTree\ a \\
replace\ p\ n\ Empty = Empty \\
replace\ p\ n\ t@(Node\ \_\ t_1\ t_2) = \textbf{if}\ p\ n\ \textbf{then}\ Node\ n\ t_1\ t_2\ \textbf{else}\ t
\end{array}
$$

Observar que si no se utilizase un @–*pattern*, sería necesario referenciar explícitamente al valor del nodo raíz de t (sea a dicho valor), y se debería escribir $Node\ a\ t_1\ t_2$ en lugar de t.

A.2.3 Polimorfismo Ad-hoc

Sea *isElem* una función que recibe una lista, un valor cualquiera cuyo tipo coincide con el de los elementos almacenados en la lista, y retorna un booleano según el valor se encuentre o no en la lista. Recurriendo al polimorfismo paramétrico (dado que aparentemente *isElem* no necesita conocer el tipo de los elementos de la lista), su definición sería:

```
isElem :: [a] → a → Bool
isElem [ ] a = False
isElem a (x : xs)
  | a == x    = True
  | otherwise = isElem a xs
```

El código de *isElem* es correcto, pero no así su tipo. Esto es debido a que *sí* existe una restricción sobre el tipo *a*. Notar que *isElem* compara sus elementos con el valor buscado mediante la función (==). Por lo tanto, *a* no puede ser un tipo cualquiera. Para que la signatura de tipo sea correcta, *a* debe ser cualquier tipo *comparable por igualdad*.

Esta modalidad de polimorfismo, donde es posible expresar restricciones sobre una variable de tipo, se conoce como polimorfismo ad-hoc [CW85]. Los lenguajes funcionales permiten declarar tales restricciones mediante *clases de tipos*. Una clase define un conjunto de operaciones, y todo tipo que instancie una clase debe proveer implementaciones para todas las operaciones de dicha clase. Por ejemplo, la clase *Eq* define la noción de tipo comparable, y todo tipo comparable debe instanciarla.

```
class Eq a where
  (==), (/=) :: a → a → Bool
  x /= y     = not (x == y)
```

Una clase admite definiciones por defecto para algunos de sus operaciones miembros, lo que significa que la definición para esta operación puede no proveerse si se acepta la implementación ya dada (en este caso, el operador de desigualdad se define como la negación del de igualdad).

La restricción para el tipo *a* de *isElem* es que debe ser instancia de la clase *Eq*. A nivel sintáctico, dicha restricción se expresa mediante el contexto ($Eq\ a \Rightarrow \ldots$), por lo que resulta $isElem :: Eq\ a \Rightarrow [a] \to a \to Bool$.

A.3 Arreglos

Haskell provee arreglos polimórficos tanto en el tipo de los elementos almacenados como en el del índice. Esto no es cierto para Clean, que provee polimorfismo sólo para los elementos. Los arreglos son uno de los puntos donde existen diferencias importantes entre Haskell y Clean, motivo por el cual serán estudiados separadamente. En esta sección sólo serán estudiados los arreglos provistos por Haskell; los arreglos Clean se discutirán en la Secc. A.4, donde se estudian los detalles específicos de dicho lenguaje.

Un arreglo Haskell se representa mediante el tipo $Array\ a\ b$, donde a es el tipo del índice y b el de los elementos. Las operaciones principales son tres: creación, lectura (dada por el operador (!)) y actualización (denotada mediante el operador (//)).

$$
\begin{array}{lll}
array & :: Ix\ a \Rightarrow (a,a) \rightarrow [(a,b)] \rightarrow Array\ a\ b \\
(!) & :: Ix\ a \Rightarrow Array\ a\ b \rightarrow a \rightarrow b \\
(//) & :: Ix\ a \Rightarrow Array\ a\ b \rightarrow [(a,b)] \rightarrow Array\ a\ b
\end{array}
$$

Un arreglo se crea dando sus límites inferior y superior mediante un par ordenado, y una lista con su contenido inicial, la cual consiste en asociaciones índice-valor. El operador (!) permite obtener un elemento de un arreglo en una posición dada. El operador (//) recibe un arreglo y una lista de asociaciones índice-valor, y retorna un arreglo actualizado en función de dicha lista. Notar el uso de polimorfismo ad-hoc en el índice, dado que no todo tipo es apto para indexar arreglos. La clase Ix fuerza a que el tipo a sea enumerable, la cual es condición indispensable para que pueda utilizarse como índice discreto.

A.3.1 El problema del *update in place*

La actualización de una posición dada de un arreglo es una operación característica y frecuentemente utilizada por los lenguajes imperativos. Mediante punteros o un operador de asignación, es posible cambiar el valor de una posición cualquiera en tiempo constante. A esta operación se la denomina *update in place*. Intuitivamente, se trata de una operación simple cuya implementación mediante un lenguaje funcional puro debería resultar trivial. Sin embargo, no es posible ejecutar dicha operación en un lenguaje funcional puro sin correr riesgos de perder la transparencia referencial. Se ilustrará el problema mediante el siguiente ejemplo.

$$
\begin{array}{l}
g \;=\; a_1 \,//\, [(i,1)] \\
\quad \textbf{where} \\
\qquad a \;\;\;\;\;\; = \; \mathit{array}\,(0,1)\,[(0,0),(1,0)] \\
\qquad (i, a_1) \;=\; (a\,!\,1,\; a \,//\, [(1,1)])
\end{array}
$$

Si se utilizara *update in place*, el arreglo retornado por la función g tendría un 1 ó un 0 en la posición 0, dependiendo de cuál de las componentes del par computado en la segunda definición del *where* se evaluase primero. Dado que el resultado de la expresión g dependería del orden de evaluación además del valor de las subexpresiones que la componen, la transparencia referencial se perdería.

El problema se soluciona eliminando el *update in place* sobre arreglos, esto es, el arreglo retornado por la operación (//) es una copia del original. Esta solución no es deseable porque la complejidad de esta operación pasa a ser $O(n)$, siendo n el tamaño del arreglo.

Debido a que los arreglos funcionales puros no admiten *update* destructivo en tiempo constante, se los llama arreglos *no mutables*. Es evidente que este tipo de arreglos no es apto para implementar algoritmos ni estructuras de datos eficientes. La próxima sección mostrará cómo Haskell logra solucionar este problema, proveyendo arreglos *mutables*.

A.3.2 Arreglos Mutables en Haskell

La clave para solucionar el problema del *update in place* deriva del motivo por el cual el falla el ejemplo dado en la sección anterior: *el arreglo a actualizar está siendo referenciado por más de una expresión.* En efecto, las dos componentes del par en la segunda definición del *where* están referenciando al arreglo. De esto modo, al ser modificado por la segunda componente se introduce un posible efecto lateral, si es que la primera componente aún no ha sido evaluada. Notar que bajo un entorno estricto no existiría este problema, dado que el orden de evaluación estaría establecido y por lo tanto, el valor del arreglo en la posición 0.

Para poder actualizar un arreglo en tiempo constante es necesario, entonces, mantener a lo largo de la ejecución de todo programa la invariancia de la siguiente propiedad: *todo arreglo mutable debe estar referenciado a lo sumo por una sola expresión.*

Haskell asegura esta propiedad mediante *mónadas* [Wad95]. Las mónadas son un concepto proveniente de la teoría de categorías, utilizadas originalmente para expresar la semántica operacional de los lenguajes imperativos.

Los detalles teóricos de las mónadas son altamente técnicos, por lo que serán evitados. La exposición se limitará a mostrar cómo se utilizan para modelar arreglos con *update* eficiente.

Informalmente, una mónada es un tipo abstracto que permite modelar computaciones secuenciables. Las mónadas son especialmente útiles para modelar computaciones basadas en el concepto imperativo de estado mutable: una computación que además de retornar un valor, modifica *in place* un estado definido ad-hoc. Este tipo de mónadas se conoce como mónadas de *estado*. Una mónada de estado permite encapsular la manipulación de un estado mutable, asegurando que a lo largo de su tiempo de vida estará referenciado por a lo sumo por una única expresión. De esta manera, es posible actualizarlo *in place* sin introducir efectos laterales. Modelando a un arreglo como parte del estado que puede ser actualizado *in place* por una computación monádica, es posible proveer arreglos mutables eficientes.

La mónada de estado provista por Haskell se denomina ST, y está definida por el tipo abstracto $ST\ s\ a$. Una expresión con dicho tipo representa a una computación que retorna un valor de tipo a, posiblemente modificando *in place* a un estado de tipo s.

Un arreglo mutable se modela mediante el tipo $STArray\ s\ i\ e$. El tipo s representa al estado mutable donde "reside" el arreglo, mientras que i y e modelan respectivamente los índices y los elementos. Las operaciones de creación, lectura y actualización pasan a ser las siguientes:

$$
\begin{aligned}
&newSTArray &&:: Ix\ i \Rightarrow (i,i) \rightarrow e \rightarrow ST\ s\ (STArray\ s\ i\ e)\\
&readSTArray &&:: Ix\ i \Rightarrow STArray\ s\ i\ e \rightarrow i \rightarrow ST\ s\ e\\
&writeSTArray &&:: Ix\ i \Rightarrow STArray\ s\ i\ e \rightarrow i \rightarrow e \rightarrow ST\ s\ ()
\end{aligned}
$$

La función *newSTArray* recibe los límites inferior y superior del arreglo, un valor de inicialización, y devuelve una computación que al ejecutarse, retornará un arreglo mutable. Dados un arreglo mutable y un índice, la función *readSTArray* devuelve una computación que retorna el elemento del arreglo correspondiente a dicho índice. Finalmente, *writeSTArray* recibe el arreglo a modificar, el índice y el nuevo valor. El resultado es una computación que modifica *in place* al estado s que modela al arreglo. Como esta computación únicamente modifica al estado sin retornar ningún valor de utilidad, se devuelve un valor de tipo (), que modela a un conjunto cuyo único elemento consiste en una 0-upla.

Haskell provee una sintaxis especial para facilitar la codificación de programas monádicos llamada *do-notation*. La idea es permitir la secuenciación de acciones monádicas con una sintaxis pseudo–imperativa. Dicha secuencia se escribe de la siguiente manera:

do
  [{]
    $action_1$ [;]
    ...
    $action_n$ [;]
    return *exp*
  [}]

Una sentencia *do* está compuesta de una o más acciones monádicas. Una acción monádica puede retornar un valor, o sólo llevar a cabo un efecto. En el primer caso, el resultado se referencia mediante la notación $res \leftarrow action$. Si una acción monádica tiene esta forma, la variable *res* puede utilizarse en cualquier otra que suceda a la que produjo el valor. La última alternativa permite construir una acción monádica a ser retornada a partir de una expresión cualquiera.

Como ejemplo se dará una función que crea un arreglo con única posición, lo actualiza, lee el nuevo valor y lo retorna.

$$
\begin{aligned}
example \;=\; & \textbf{do} \\
& \quad a \leftarrow newSTArray\ (0,0)\ 0 \\
& \quad writeSTArray\ a\ 0\ 1 \\
& \quad v \leftarrow readSTArray\ a\ 0 \\
& \quad \textbf{return}\ v
\end{aligned}
$$

Notar que la mónada *ST* oculta el estado donde reside el arreglo mutable, haciéndolo accesible sólo mediante operaciones que fuerzan a utilizar el arreglo de manera secuencial (en el ejemplo primero se crea, *luego* se actualiza, y *luego* se accede). De este modo se impide crear referencias múltiples dentro de una misma acción monádica, evitándose la introducción de efectos laterales.

A.4 Introducción a Clean

Los conceptos estudiados a lo largo de este apéndice (salvo por la Secc. A.3) son aplicables también a Clean, a pesar de estar ejemplificados en Haskell.

No obstante, Clean presenta diferencias importantes con respecto a Haskell en dos aspectos: el sistema de tipos y la implementación de arreglos. Esta sección estudia esas diferencias.

A.4.1 Sistema de Tipos

El sistema de tipos de Clean posee todas las características enumeradas en la Secc. A.2, incluyendo polimorfismo ad-hoc mediante clases de tipos. Además, Clean provee dos extensiones: anotaciones de *strictness* y *unique types*.

Anotaciones de *Strictness*

Las anotaciones de *strictness* permiten declarar una expresión como estricta, esto es, dejar de lado la regla "computar sólo si es necesario" forzada por la *lazy evaluation* y computar dicha expresión. Dichas anotaciones se denotan mediante el operador (!). Se ilustrará su uso mediante las funciones *iterate* y *stIterate*. Notar que las signaturas de tipo son ligeramente distintas a las de Haskell, además de las anotaciones de *strictness*: el símbolo ($\rightarrow$) es único y separa los parámetros (separados entre sí mediante espacios) del resultado.

$$
\begin{array}{l}
iterate\ ::\ (a\ \rightarrow\ a)\ a\ \rightarrow\ a \\
iterate\ f\ x\ =\ [x\ :\ iterate\ f\ (f\ x)]
\end{array}
$$

$$
\begin{array}{l}
stIterate\ ::\ (a\ \rightarrow\ a)\ a\ \rightarrow\ [!a] \\
stIterate\ f\ x\ =\ [x\ :\ stIterate\ f\ (f\ x)]
\end{array}
$$

$$
\begin{array}{ll}
g\ n\ =\ length\ (take\ n\ (stIterate\ succ\ 0)) & \text{— mal uso de } stIterate \\
h\ n\ =\ sum\ (take\ n\ (stIterate\ succ\ 0)) & \text{— buen uso de } stIterate
\end{array}
$$

Al recibir una función f y un valor inicial x, *iterate* y *stIterate* computan la secuencia infinita $[x, fx, f(fx), \ldots]$. Si bien el código de ambas funciones es el mismo, hay una diferencia importante: el tipo de los elementos de la lista producida por *stIterate* está declarado estricto. Por lo tanto, esta función produce una lista de valores ya computados, mientras que *iterate* produce una lista con elementos *lazy*, que serán computados sólo cuando sean utilizados.

Las anotaciones de *strictness* son muy importantes, pues permiten controlar el orden de evaluación de manera flexible y sencilla, lo cual da una forma indirecta de controlar en parte el uso de memoria. Esto puede verse en la

cantidad de memoria utilizada para almacenar una lista de tipo $[a]$ comparado con la utilizada por una de tipo $[!a]$. La primera almacena computaciones diferidas (*closures*) que permitirán calcular el valor correspondiente, por lo que todos los datos y porciones de tal computación deben ser almacenados; en cambio, la segunda sólo almacena los resultados, lo cual ocupa, en general, mucho menos espacio que los *closures* que calculan dichos datos. Asumiendo que un entero ocupa 2 bytes y un *closure* que computa un entero $2k$ bytes, $k \in \mathbb{Z}$, una lista de *closures* ocupará k veces más espacio, lo que puede ser muy costoso si la lista es extensa.

Debe notarse que no siempre el uso de anotaciones de *strictness* resulta conveniente. La función g muestra un uso indebido de este mecanismo: g sólo computa la longitud de un segmento inicial de la lista, para lo que no hace falta el valor de los elementos almacenados, que son evaluados innecesariamente. A su vez, las anotaciones de *strictness* son beneficiosas en la función h, que sí utiliza a los elementos de dicho segmento inicial para computar su sumatoria. Luego, si la lista tiene muchos elementos y todos ellos serán utilizados, es mejor usar la alternativa estricta, ya que computar los elementos no supone un costo adicional y el espacio disminuye en un factor constante.

Unique Types

Una expresión con tipo *unique* [SBvEP94, Kes94] sólo puede tener a lo sumo una referencia a lo largo de su tiempo de vida dentro de un programa. Esto es, sólo puede estar referenciado por a lo sumo una sola expresión. En el siguiente ejemplo, *avg* calcula el promedio de una lista creando dos referencias sobre su argumento l; por lo tanto, l no es *unique*. En cambio, *avgU* referencia una sola vez a l (en la expresión *sumlen* l), por lo que l es *unique* en esta función. Notar también que *avgU* es más eficiente que *avg*, ya que recorre la lista sólo una vez. En cambio, *avg* la recorre dos veces, una para computar la sumatoria de sus elementos y otra para obtener su longitud.

```
avg :: [Float] → Float    — promedio, l no es unique
avg l = sum l / length l

avgU :: *[Float] → Float    — promedio, l es unique
avgU l = s / n
  where
    (s, n)          = sumlen l
    sumlen []       = (0, 0)
    sumlen [x : xs] = let (s, n) = sumlen xs in (x + s, n + 1)
```

Clean traslada al nivel de chequeo de tipos la verificación de la invariancia de la propiedad de "referencia única". Esto es, si l hubiese sido declarada *unique* en la signatura de tipo de *avg*, el programa no hubiese compilado. En cambio, es correcto declarar *unique* a l en *avgU*.

Existen dos mecanismos para declarar unicidad. El operador $(*)$ fuerza a que un tipo sea *unique*. Esto es, para todo tipo a, $*a$ es la variante *unique* de a (p. ej., $*[Float]$ en la función *avgU*). La otra manera de especificar unicidad es utilizar el operador (.), que denota *posible* unicidad. Sea $f :: .a \rightarrow .b$; si f se invoca con un parámetro *unique*, entonces todo tipo precedido por (.) se considera *unique*, por lo que el valor retornado también lo es. Igualmente, si se invoca a f con un argumento no *unique*, tampoco lo será $.b$, por lo que el resultado de la aplicación no es *unique*.

La experiencia muestra que preservar la unicidad no es una tarea sencilla, dado que es fácil escribir código que duplique referencias sobre una expresión dada. Por ejemplo, la siguiente función provoca un error de tipo al ser compilada. El motivo es que las dos componentes del par retornado no son *unique*, ya que ambas referencian al parámetro recibido.

$$wrong :: *Int \rightarrow (*Int, *Int)$$
$$wrong\ n = (n + 1,\ n - 1)$$

Si bien en este ejemplo la causa de la pérdida de unicidad es evidente, existen otros donde el motivo es mucho más sutil y difícil de detectar. La verdadera utilidad de este mecanismo se verá en la próxima sección, donde se estudia cómo Clean saca provecho de los *unique types* para proveer arreglos mutables.

A.4.2 Arreglos Mutables en Clean

Los *unique types* aseguran, vía chequeo de tipos, que un programa está codificado de modo que preserve la unicidad de los elementos de un tipo así declarado. De este modo, es posible actualizar *in place* a un arreglo *unique* sin poner en peligro la transparencia referencial.

Un arreglo de elementos con tipo a se declara como $\{a\}$. Luego, $*\{a\}$ es un arreglo *unique*, que puede ser modificado en tiempo constante. A su vez, $.\{a\}$ declara un arreglo posiblemente *unique*.

Las operaciones sobre arreglos provistas por Clean pueden clasificarse en dos categorías: las que operan exclusivamente sobre arreglos mutables, y las que funcionan sobre ambos tipos de arreglos. A continuación se listan las operaciones principales.

$$
\begin{array}{ll}
\mathit{createArray} & :: !\mathit{Int} \;\; e \rightarrow .\{e\} \\
\mathit{select} & :: !.\{e\} \;\; !\mathit{Int} \rightarrow e \\
\mathit{uselect} & :: !*\{e\} \;\; !\mathit{Int} \rightarrow (e, !*\{e\}) \\
\mathit{update} & :: !*\{.e\} \;\; !\mathit{Int} \;\; .e \rightarrow .\{.e\} \\
\mathit{replace} & :: !*\{.e\} \;\; !\mathit{Int} \;\; .e \rightarrow (.e, !*\{.e\})
\end{array}
$$

La función *createArray* crea un arreglo, dado su tamaño inicial y un elemento de inicialización; el arreglo resultante es posiblemente *unique*: el compilador resolverá su unicidad en función de cómo se lo utilice en el resto del programa. La función *select* permite seleccionar un elemento de un arreglo posiblemente *unique*. Es de destacar que esta función anula la unicidad del arreglo recibido, ya que agrega una referencia: si el arreglo tenía una sola, ahora pasa a tener dos, ya que *select* también lo referencia. En efecto, *select a i* referencia al arreglo *a*, por lo tanto, cualquier referencia posterior no puede tratar al arreglo *a* como *unique*. Para poder obtener elementos de un arreglo sin violar su unicidad debe utilizarse *uselect*, la cual retorna además del elemento deseado, una *nueva* versión del arreglo, sin referenciar. De este modo, es posible seguir utilizando como *unique* este nuevo arreglo. Notar que el elemento del arreglo retornado por *uselect* (y también por *select*) no puede ser *unique*, dado que ya está referenciado por el arreglo donde se almacena. La función *update* permite actualizar *in place* una posición dada de un arreglo mutable. La unicidad del arreglo resultante está definida por la unicidad del nuevo elemento: si hay más de una referencia a algún elemento de un arreglo, no puede ser considerado *unique*. La función *replace* permite seleccionar un elemento de un arreglo mutable sin alterar su posible atributo de unicidad. Esto se logra dando un elemento que reemplazará en el arreglo al ya seleccionado. De este modo, no aumenta la cantidad de referencias al elemento original, ya que dejó de ser referenciado por el arreglo que lo contenía. Esta función se utiliza en lugar de *select* y *uselect* para seleccionar elementos *unique* preservando dicho atributo.

Como ejemplo se dará la función *modify*, que dada una matriz mutable y un par de índices, le aplica una función al elemento ubicado en esa posición, actualizando la matriz *in place* con el nuevo valor. Si bien el ejemplo es sencillo, ilustra los cuidados que deben tenerse al trabajar con estructuras *unique* cuyos elementos también lo son (una matriz *unique* es un arreglo *unique* de arreglos también *unique*). El problema es que al obtener una fila de la matriz, esta pierde su unicidad, ya que todavía está siendo referenciada por la matriz. Por lo tanto, deber ser *reemplazada* por un nuevo valor mientras

es utilizada por la función que la accedió. Finalmente, dicha función deberá actualizar la matriz, reemplazando la fila temporal por el valor resultante de aplicar la función.

$$
\begin{array}{l}
modify :: *\{*\{*a\}\}\ (Int, Int)\ (.a \rightarrow .a) \rightarrow *\{*\{*a\}\} \\
modify\ m\ (i,j)\ f\ = \\
\quad \textbf{let!} \\
\qquad (row, m_1) = replace\ m\ i\ \{\} \\
\qquad (v, row_1) = uselect\ row\ j \\
\qquad row_2 = update\ row_1\ j\ (f\ v) \\
\quad \textbf{in} \\
\qquad update\ m_1\ i\ row_2
\end{array}
$$

La función *modify* necesita ejecutar cuatro operaciones sobre la matriz: un reemplazo, una selección, y dos actualizaciones. En Haskell, la misma función puede implementarse en sólo dos pasos: obtener el elemento de la matriz (un arreglo indexado por pares ordenados), y actualizarla con el nuevo valor:

$$
\begin{array}{l}
modify :: Ix\ i \Rightarrow STArray\ s\ (i,i)\ a \rightarrow (i,i) \rightarrow (a \rightarrow a) \rightarrow ST\ s\ () \\
modify\ m\ pos\ f\ = \\
\textbf{do} \\
\quad v \leftarrow readSTArray\ m\ pos \\
\quad writeSTArray\ m\ pos\ (f\ v)
\end{array}
$$

Mediante este ejemplo puede apreciarse que preservar la unicidad puede ser una tarea complicada, lo cual es una desventaja importante de los arreglos mutables Clean con respecto a los provistos por Haskell. Esto se debe a que las mónadas proveen un mecanismo para *asegurar* unicidad mediante encapsulamiento del arreglo mutable; por el contrario, Clean provee simplemente una manera de *verificar* mediante chequeo de tipos que un arreglo pueda ser o no actualizado *in place*. Por lo tanto, Clean deja en manos del programador la responsabilidad de implementar funciones que operen sobre *unique types* correctamente, de modo que no violen la propiedad de unicidad.

A.5 Conclusiones

Se mostraron las características de los lenguajes funcionales puros utilizados a lo largo de esta tesis. Como se dijo al comienzo del apéndice, la intención es únicamente la de refrescar estas nociones y no la de proveer material de estudio, por lo que las ideas presentadas son necesariamente escuetas e incompletas.

Se reitera una vez más que aquellos lectores interesadas en la programación funcional consulten material bibliográfico especializado [Dav92, Tho96, Bir98], u otros.

Bibliografía

[AGH01] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison–Wesley, 3rd. edition, June 2001.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison–Wesley, 1983.

[AS89] C. Aragon and R. Seidel. Randomized search trees. *Algorithmica*, 16(4/5):464–497, October 1989.

[BC01] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. Linux Series. O'Reilly Associates, first edition, January 2001.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice–Hall Press, 2nd. edition, 1998.

[BMV99] J. Blanco, P. Mocciola, and D. Vilela. Verificación de programas funcionales imperativos y concurrentes. In *Anales del tercer Workshop Argentino de Informática Teórica (WAIT'99)*, Buenos Aires, Argentina, 6–7 September 1999.

[BR95] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice & Experience*, 25(8):863–889, Aug 1995.

[BS92] E. Barendsen and J.E.W. Smetsers. Graph rewriting and copying. Technical report, University of Nijmegen, August 1992.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Dav92] Antony J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992.

[Dev98] Luc Devroye. Branching processes and their applications in the analysis of tree structures and tree algorithms. In M. Habib, C. McDiarmid, et al., editors, *Proceedings of Implementation of Functional Languages*, volume 16 of *Algorithms and Combinatorics*, pages 249–314. Springer–Verlag, 1998.

[Dev99] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Duxbury Press, 5th edition, December 1999.

[DR95] Luc Devroye and Bruce Reed. On the variance of height of random binary search trees. *SIAM Journal on Computing*, 24(6):1157–1162, December 1995.

[GR83] A. Goldberg and D. Robson. *Smalltalk 80. The Language and its implementation*. Addison–Wesley, 1983.

[Gro97] John H. G. van Groningen. The implementation and efficiency of arrays in Clean 1.1. In W. Kluge, editor, *Proceedings of Implementation of Functional Languages, 8th International Workshop, IFL '96*, volume 1268 of *LNCS*, pages 105–124. Springer–Verlag, 1997.

[Gue97] Laurent Guerby. *Ada 95 Rationale: The Language – The Standard Libraries*, volume 1247 of *LNCS*. Springer–Verlag, July 1997.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programing. *Comunications of the ACM*, 12:576–580, 1969.

[Hud92] P. Hudak. Mutable abstract datatypes or how to have your state and munge it too. DCS RR–914, Yale University, New Haven, Connecticut, USA, 1992.

[Hug89] John Hughes. Why functional programming matters? *Computer Journal*, 32(2):98–107, 1989.

[Inc01] Sun Microsystems Inc. *The Java Hotspot Virtual Machine*. Sun Press, 2001.

[Jon95] Mark Jones. A system of constructor classes: Overloading and implicit higher–order polymorphism. *Journal of Functional Programming*, 5(1):1–31, October 1995.

[Kes94] M. Kesseler. Uniqueness and lazy graph copying – copyright for the unique. In *Proceedings of the 6th International Workshop on the Implementation of Functional Languages*, University of East Anglia, Norwich, UK, 1994.

[KR88] B. W. Kernigan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd. edition, June 1988.

[KW92] David King and Philip Wadler. Combining monads. In *Glasgow Workshop on Functional Programming*. Workshops in Computing, Springer–Verlag, July 1992.

[Lau93] J. Launchbury. Lazy imperative programming. In *ACM SIGPLAN Workshop on State in Programming Languages*, pages 46–56. ACM Press, 1993.

[L'E88] P. L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6), June 1988.

[LPJ94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 24–35, Orlando, Florida, 20–24 June 1994. *SIGPLAN Notices* 29(6), June 1994.

[LPJ95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Journal of Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[ML98] Pablo E. Martínez López. Application of functional languages to massive and complex computational problems. Master thesis, PEDECIBA, Universidad de La República, Montevideo, Uruguay, September 1998.

[Oka95] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[Oka96a] Chris Okasaki. Functional data structures. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 131–158. Springer–Verlag, August 1996.

[Oka96b] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, September 1996.

[Oka99] Chris Okasaki. *Edison User's Guide (Haskell Version)*, May 1999. URL: `http://www.haskell.org/ghc/docs/edison`.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice–Hall, May 1987.

[PJ97] Simon L. Peyton Jones. Bulk types with class. In *Proceedings of the Haskell Workshop*, Amsterdam, June 1997.

[PJH99] Simon L. Peyton Jones and John Hughes (editors). Haskell 98: A non–strict, purely functional language, February 1999. URL: `http://www/haskell.org/onlinereport`.

[PJHH+93] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phillip Wadler. The Glasgow Haskell compiler: A technical overview. In *Join Framework for Information Technology Technical Conference*, Keele, 1993.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (PoPL)*, pages 71–84, Charlotte, North Carolina, January 1993.

[PMP95] Pedro Palao, Manuel Núñez, and Ricardo Peña. A second year course on data structures based on functional programming. In P. P. Hartel and R. Plasmeijer, editors, *Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 65–84. Springer–Verlag, December 1995.

[PvE98] Rinus Plasmeijer and Marko van Eekelen. The Concurrent Clean language report. Version 1.3. Technical report, High Level Software Tools B.V. and the University of Nijmegen, 1998.

[Röj94] Niklas Röjemo. `nhc`: a space–efficient Haskell compiler. In *Proceedings of the Workshop on Implementation of Functional Languages*, pages 30.1–30.29, Norwich, September 1994.

[SBvEP94] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In H. Schneider and H. Ehrig, editors, *Proceedings of Graph Transformations in Computers Science, International Workshop*, volume 776 of *LNCS*, pages 358–379. Springer–Verlag, 1994.

[SF95] R. Sedgewick and P. Flajolet. *An introduction to the Analysis of Algorithms*. Addison–Wesley, November 1995.

[SL95] A. Stepanov and M. Lee. The standard template library. Technical report, Silicon Graphics Inc., Hewlett–Packard Laboratories, October 1995.

[SPJ95] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non–strict functional languages. *22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, January 1995.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison–Wesley, 1994.

[Tho96] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison–Wesley, 1996.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zulstra. Design and implementation of an efficient priority queue. In *Mathematical Systems Theory*, volume 10, pages 99–127. Springer–Verlag, 1977.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[Wad92a] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. (Special issue of selected papers from 6'th Conference on Lisp and Functional Programming).

[Wad92b] Philip Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[Wad93] Philip Wadler. A taste of linear logic (invited talk). In *Mathematical Foundations of Computer Science*, volume 711 of *LNCS*. Springer–Verlag, August 1993.

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 65–84. Springer–Verlag, May 1995.