Análisis asintótico amortizado en lenguajes funcionales perezosos

Carlos Gonzalía [1]
GIIA - Grupo de Investigación en Inteligencia Artificial
Departamento de Ciencias de la Computación
UNS - Universidad Nacional del Sur
Bahía Blanca, Argentina.

Septiembre de 1997

Resumen

Los lenguajes funcionales con régimen de evaluación perezosa presentan dificultades para el análisis de la complejidad asintótica de los algoritmos programados en ellos. Las técnicas de análisis amortizado permiten en muchos casos análisis accesibles y útiles sobre el comportamiento de una estructura de datos en dichos lenguajes. En este trabajo se da un panorama de estas cuestiones y se analiza la estructura de datos de montículo sesgado implementada en Haskell, como un ejemplo de la utilidad de dichas técnicas para el programador funcional.

[1]Dirección: Dpto. de Cs. de la Computación, Avenida L.N. Alem 1253, 8000-Bahía Blanca, Prov. de Bs. As., Argentina. E-mail: gonzalia@criba.edu.ar. Teléfono: 054-091-41992

Análisis asintótico amortizado en lenguajes funcionales perezosos

1 Introducción

En el debate tradicional dentro del paradigma funcional acerca de evaluación estricta versus evaluación perezosa, uno de los puntos importantes es la gran dificultad de los lenguajes perezosos a la hora de evaluar la *complejidad asintótica* de los algoritmos programados en ellos, en tanto que los lenguajes estrictos admiten una aplicación más directa de las técnicas estándar de análisis. Sin embargo, el grado de abstracción y modularidad permitido por la evaluación perezosa hacen preferible a este régimen de ejecución ([Hug89]), por lo que se hace imprescindible encontrar formas razonables de encarar un análisis de complejidad en dichas condiciones. En los últimos años se ha propuesto con éxito la aplicación de nuevas versiones de *análisis amortizado* a estructuras de datos en lenguajes perezosos, permitiendo que el programador funcional pueda dar cotas útiles al tiempo de ejecución de sus algoritmos.

La sección 2 presenta los conceptos del análisis amortizado tal y como se originan en el paradigma imperativo, y muestra cuáles son los requisitos para que tal análisis sea posible. La sección 3 indica los problemas que tiene la aplicación directa de esta técnica, y desarrolla las modificaciones necesarias a dichos métodos de análisis amortizado para tener en cuenta un régimen de evaluación perezosa. La sección 4 muestra un ejemplo de análisis amortizado bajo evaluación perezosa, usando como ilustración los montículos sesgados. Finalmente, la sección 5 menciona algunos puntos de interés que el análisis amortizado tiene para el programador funcional. El presente trabajo no tiene la intención de introducir nuevos aportes al análisis amortizado de programas funcionales, sino dar una visión integradora de los aportes conocidos y un ejemplo detallado del uso de esta técnica, para ilustrar tanto la forma en que se lleva a cabo como las ventajas que un programador funcional obtiene con dicho análisis.

2 Análisis amortizado

El *análisis amortizado* es una técnica de análisis de complejidad asintótica de algoritmos originada dentro del paradigma imperativo hace no muchos años. Se basa en la idea de considerar el costo, en el peor caso, de ejecutar una sucesión de ope-

raciones, en vez del tradicional análisis (también en el peor caso) de una operación individual. La razón de fondo para esta distinción es que los algoritmos ejecutan sucesiones de operaciones, y dichas operaciones suelen exhibir fuertes correlaciones entre sí, de modo que el tiempo de ejecución en el peor caso de una tal sucesión suele ser menor que la suma de las cotas de las operaciones individuales, haciendo a las cotas convencionales demasiado pesimistas. Al ser un análisis amortizado un análisis en el peor caso, buscamos una cota a la más costosa sucesión de operaciones imaginable.

Por ejemplo, si tal análisis determina que el costo amortizado de una sucesión de M operaciones es de $O(M \log n)$ en el peor caso, entonces podemos pensar en un sentido puramente formal que cada operación individual toma $O(\log n)$. De esta manera, podemos calcular el costo de cualquier sucesión de operaciones multiplicando esta cota amortizada por el número de operaciones involucradas. Las cotas amortizadas son más débiles que las cotas individuales de peor caso, ya que no dan una garantía del máximo tiempo que podría tomar una operación individual. Por otra parte, son cotas más fuertes que las cotas individuales de caso promedio, ya que estas cotas promedio pueden degenerar repetidamente en costos mayores, haciendo que la sucesión de operaciones tome más tiempo que el costo total esperado.

En el contexto del paradigma imperativo, el análisis amortizado se ha usado con éxito para simplificar estructuras de datos complejas, preservando al mismo tiempo el orden del costo, ahora expresado como una cota amortizada. Un ejemplo claro de esto es el problema de los árboles binarios de búsqueda. La aproximación ingenua permite que los árboles degeneren en su forma, haciendo de $O(n)$ al tiempo de buscar un elemento. La solución tradicional es hacer que las operaciones garantizen el balance del árbol, complicando la estructura y el algoritmo de manipulación, dando los conocidos árboles AVL, donde la operación de búsqueda es de costo $O(\log n)$ en el peor caso. Sin embargo, si se persigue una cota amortizada de la misma complejidad, el uso de los llamados *árboles splay* consigue el mismo orden, pero amortizado ([Mor90]). Si bien una operación individual puede seguir costando $O(n)$, la estructura de datos impide que esto suceda repetidamente, por lo que el costo amortizado de una sucesión de M operaciones es de $O(M \log n)$, y tanto la representación como la manipulación resultan simplificadas.

En un contexto imperativo, la complejidad amortizada está íntimamente relacionada con las estructuras de datos *autoadaptables*, las cuales cambian su forma no sólo en respuesta a las operaciones que naturalmente esperamos tengan ese efecto, sino también como un efecto secundario a operaciones que no esperamos que lo tengan. En este sentido, los árboles splay son autoadaptables según la sucesión de

operaciones de búsqueda realizadas. En nuestro contexto funcional, tal capacidad de autoadaptación es igualmente útil para garantizar órdenes de costo razonables en los algoritmos. También es importante notar que las operaciones de tal sucesión bien pueden (y suelen) ser de distinto tipo, por lo que la técnica se adapta muy adecuadamente al análisis de un tipo abstracto de datos como objeto aislado.

Los métodos clásicos de análisis amortizado se basan en el hecho de que uno puede tener una operación individual costosa, a condición de poder repartir dicho costo entre todas las operaciones baratas de la sucesión. Una analogía útil para pensar en esto es considerar que a cada operación de la sucesión se le concede su tiempo amortizado para ser resuelta. Si una operación tarda menos que este tiempo, el exceso no gastado se ahorra para el futuro. Si una operación tarda más que su tiempo amortizado, recurrirá a dichos ahorros previos para equilibrar el costo sobre la sucesión ([Wei94]). Para que esto sea posible, la estructura de datos debe comportarse de una manera *single-threaded*, ya que esos ahorros obtenidos en el pasado sólo podrán ser gastados una única vez en el futuro. Una estructura de datos *multi-threaded* es tal que sus múltiples futuros pueden necesitar de dichos ahorros más de una vez, lo que es imposible y destruiría la cota amortizada.

En el caso de los lenguajes imperativos, el escenario de ahorros puede ser usado de dos maneras distintas, dependiendo de la manera en que los mismos sean distribuídos. En el el llamado *método del banquero*, dichos ahorros se piensan como créditos, los cuales son asociados con posiciones individuales dentro de la estructura de datos, y se usan para pagar el acceso futuro a esas mismas posiciones. En el *método del físico*, los ahorros están asociados a la estructura como un todo, y ese todo se considera un potencial que aumenta y disminuye de acuerdo al costo de las operaciones realizadas ([Wei94]). El método del físico es más simple de aplicar, pero no tiene un arco de aplicación tan grande como el método del banquero. En ambos métodos, para demostrar una cota amortizada uno debe probar que, para la peor sucesión de instancias de la operación bajo análisis, los ahorros equilibran finalmente a los costos excesivos, dando una cota promedio por operación igual a la deseada.

3 Elección de una amortización adecuada

Esta idea de mantener ahorros, necesaria para poder encarar un análisis amortizado, no es aplicable en un contexto funcional tal y como es usada en algoritmos imperativos. El problema está en que las estructuras de datos de un lenguaje funcional son por naturaleza *persistentes*, es decir, luego de una actualización la vieja estructura es aún accesible. Esto permitiría que una operación individual costosa sea repetida

muchas veces, imposibilitando aprovecharse de cualesquiera ahorros pasados. Lo que se necesita es que, luego de resolver la primera de dichas operaciones individuales costosas, el resultado calculado quedará disponible para el futuro, es decir memoizado y compartido. En lenguajes con un régimen de evaluación perezosa, estas propiedades están presentes y nos permitirán rescatar la idea de análisis amortizado una vez que resolvamos el problema de la persistencia.

La forma de solucionar esta incompatibilidad es cambiar de punto de vista, y en vez de pensar en ahorrar créditos para uso futuro, hablamos de *deudas* a pagar. Estas deudas se crean cuando la evaluación perezosa suspende el cómputo, y se van pagando a medida que posteriores operaciones vayan accediendo a las partes aún no calculadas de dicha estructura. Como no hace ningún daño pagar más de una vez una deuda, tenemos ahora una posibilidad de compatibilizar la amortización con los múltiples futuros: permitir que los múltiples futuros paguen cada uno la misma deuda ([Oka96]).

El tipo de amortización necesaria consiste entonces en crear, para cada cómputo demorado por la evaluación perezosa, una deuda proporcional a su costo efectivo eventual. A cada operación le asignamos una cantidad de créditos que pueden ser usados para pagar deudas, y que intuitivamente hacemos corresponder con la necesidad de utilizar un resultado demorado (en parte o en su totalidad) dentro de dicha operación. Si logramos ahora mostrar que, dada una cierta asignación de créditos, todo débito será saldado para el momento en que su cómputo demorado sea necesario, habremos probado una cota amortizada. Nótese que los créditos superfluos no son "ahorrados", y que saldar débitos es simplemente un esquema de contabilidad de costos, no significando la ejecución efectiva del cómputo demorado. La prueba de la cota debe mostrar que el pago de las deudas precede a dicha ejecución.

De esta manera, podemos mostrar que cualquier thread individual termina pagando sus deudas para el momento en que el resultado es necesario, y entonces evidentemente el primer thread que fuerce un cómputo demorado pagará dicha deuda. Es claro que esta aproximación resuelve el problema, y además lo hace de una manera muy conveniente, ya que nos evita el analizar las dependencias entre los distintos threads. El hecho de "pagar una deuda más de una vez" corresponde simplemente a acceder a una parte de la estructura de datos que ya fue computada, y que permanece accesible en virtud del mecanismo de llamada por necesidad. En realidad, el peor caso ha pasado a ser ahora el tener un único futuro, desperdiciando así la memoización.

Para nuestro contexto funcional, el *método del banquero*, modificado según la perspectiva de deudas para los cómputos demorados, crea para dichos cómputos

una cantidad de débitos proporcional a su costo eventual real, y asocia cada débito con una posición dentro de la estructura de datos. La elección de dichas posiciones es totalmente dependiente del problema, siendo importante el hecho de ser el cómputo *monolítico* o *incremental*. Cada operación puede pagar una cantidad de débitos proporcional a su costo amortizado, y el orden en que los débitos son cancelados es también dependiente del cómputo. Para mostrar una cota amortizada, debemos mostrar que, cuando se accede a una posición de la estructura y posiblemente se retome un cómputo demorado, todos los débitos asociados con dicha posición ya han sido pagados. Como parte de esta demostración, es común establecer alguna clase de invariante que implica a la cota amortizada. Es evidente que las funciones incrementales son muy compatibles con este tipo de análisis amortizado, al permitir acceder a una posición una vez se han pagado sus débitos independientemente de los débitos que puedan haber en otras posiciones, lo que no es posible en una función monolítica (en la que se debe prever el haber pagado todo débito para el momento en que se necesite el resultado).

En el caso del *método del físico*, en un contexto imperativo se basa en una función que asocia a cada estructura de datos un potencial, el cual es una cota inferior al ahorro total producido por la sucesión de operaciones efectuadas. Para transformar esto a nuestro punto de vista de débitos, usamos una función que asocia a cada estructura de datos un potencial representando una cota superior de la deuda pendiente debido a los cómputos demorados. El costo amortizado se define ahora como el costo real de los cómputos demorados menos el cambio en potencial. Para mostrar una cota amortizada en este contexto, se debe mostrar que la totalidad de la deuda de la estructura de datos ha sido pagada antes de que necesitemos el resultado de cualquier parte de dicha estructura. En consecuencia, este método se adapta bien a las funciones monolíticas, para las cuales podemos predecir cuando tendrán lugar las operaciones costosas.

4 Un ejemplo con colas con prioridades

Es bien conocida la estructura de datos llamada de cola con prioridades, la cual está provista de operaciones de inserción de un elemento y recuperación del mínimo elemento. Una de las implementaciones más comunes de estas colas es por medio de la estructura de *montículo*, que suele utilizar un arreglo simple de los elementos, aprovechándose de la propiedad estructural de árbol binario completo. En estas condiciones, mezclar dos colas para producir una nueva representa un costo de $O(n)$, ya que debemos proceder por inserción repetida de los elementos de una cola en la

otra. Es claro que si esperamos mejorar el tiempo de la mezcla, deberemos usar estructuras de base distintas de los arreglos, y recurrir a punteros (ya sea físicos o conceptuales, como está implícito en las listas funcionales) ([Wei94]).

Una estructura que ha sido propuesta para resolver eficientemente la mezcla de colas es el llamado *montículo izquierdista*. Esta clase de montículos reciben su nombre del hecho de que tratan de mantener una estructura de árbol lo más inclinada hacia la izquierda que sea posible, para facilitar la mezcla (que se basa en la mezcla de la rama más a la derecha de los árboles). Esto requiere almacenar junto con cada nodo información acerca de la estructura del árbol, en forma muy similar a los árboles AVL. Así también, los algoritmos de manipulación de esta estructura son más complejos que los de los montículos simples. Es bien conocido que el tiempo en el peor caso de mezclar dos montículos izquierdistas es de $O(\log n)$ ([Wei94]). La inserción se resuelve con una mezcla, tomando al nuevo elemento como una cola unitaria, y la eliminación del mínimo procede mezclando los dos subárboles de la raíz considerados como colas.

La estructura de *montículo sesgado* (*skew heap*) es una versión autoajustable del montículo izquierdista. Como es habitual en las estructuras que apuntan a cotas de peor caso, los montículos izquierdistas son mucho más complejos de representar y manipular que los montículos normales. Los montículos sesgados logran la misma eficiencia para la mezcla con una representación y manipulación mucho más simples, y son un caso sencillo e interesante para ilustrar las técnicas de análisis amortizado en un contexto funcional. Esta estructura fue analizada bajo amortización en [Kal91] y [Sch97], pero para un contexto funcional sin persistencia, lo que hace a dicho análisis limitativo para los propósitos de un programador en un lenguaje con evaluación perezosa. Además, dicho trabajo tiene por meta la *derivación* de una cota sumamente ajustada para este tipo de montículos, por lo cual el análisis es (naturalmente) mucho más complejo que el que mostraremos aquí, ya que no estamos interesados en resultados tan ajustados.

Un montículo sesgado es estucturalmente un árbol binario, al que se le añade una propiedad de orden de montículo, de modo que el valor en cualquiera de sus nodos es siempre menor o igual al valor en sus dos hijos. La diferencia con un montículo normal es que la estructura admite una operación de mezcla con otro montículo sesgado de una manera eficiente. Para lograr esto de modo de amortizar el costo, la mezcla de dos montículos sesgados procede así: se mezclan los caminos derechos (los que se obtienen desde la raíz siempre tomando el hijo derecho), y el resultado pasa a ser el nuevo camino izquierdo (el análogo al camino derecho pero tomando siempre el hijo izquierdo); para cada nodo en este nuevo camino, excepto el último,

el anterior subárbol izquierdo se conecta como nuevo subárbol derecho. Es claro que el tiempo de realizar una mezcla tal es proporcional a la suma de las cantidades de nodos en los caminos derechos de los montículos mezclados, y potencialmente lineal si los árboles degeneran en forma. Sin embargo, se puede mostrar que en un contexto imperativo la mezcla tiene un costo amortizado de $O(\log n)$ ([Wei94]).

El siguiente es un código *Haskell* ([Has97]) implementando este tipo de datos:

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

merge :: (Ord a) => BinTree a -> BinTree a -> BinTree a
merge Empty Empty               =  Empty
merge Empty t@(Node x l Empty)  =  t
merge Empty (Node x l r)        =  Node x (merge Empty r) l
merge t Empty                   =  merge Empty t
merge t1@(Node x1 l1 r1) t2@(Node x2 l2 r2)
                    | x1<=x2    =  Node x1 (merge r1 t2) l1
                    | otherwise =  Node x2 (merge r2 t1) l2

insert :: (Ord a) => a -> BinTree a -> BinTree a
insert x t                      = merge t (Node x Empty Empty)

deleteMin :: (Ord a) => BinTree a -> (a, BinTree a)
deleteMin (Node x l r)          = (x, merge l r)
```

Como es evidente a partir de este código, las operaciones de insertar un elemento o borrar el mínimo elemento de la cola representada por medio del montículo sesgado, son nada más que usos especiales de la operación de mezcla. En consecuencia, el análisis de dicha operación nos dará una cota amortizada sobre cualquier sucesión de operaciones realizadas sobre un montículo sesgado.

Si bien `merge` es claramente incremental, como los únicos accesos realizados a la estructura de datos son a la raíz del montículo, podemos asociar las deudas y sus pagos a dicha raíz, o equivalentemente, a la estructura como un todo. Estamos entonces en condiciones de aplicar el método del fisico, simplificando el análisis. Como suele suceder, la elección de una función de potencial adecuada puede ser un asunto trabajoso. En nuestro caso, dicha función debe acotar la deuda incurrida, y una elección útil es considerar el número de nodos que, en algún sentido, no tengan un camino derecho extenso. Cuando el camino derecho de todo el montículo tenga gran cantidad de nodos en estas condiciones, será porque hemos realizado el suficiente trabajo para volcar el árbol hacia la izquierda. Formalizando el concepto:

un nodo p es *pesado* si el número de descendientes del subárbol derecho de p es al menos la mitad del número de descendientes de p, y es *liviano* en caso contrario (notar que un nodo es descendiente de sí mismo). La función de potencial será entonces el número de nodos livianos en el camino derecho del montículo.

Debemos probar que el costo amortizado de `merge` es de $O(\log n)$. Sean dos montículos, uno de un total de n_1 nodos y con l_1 nodos livianos y h_1 nodos pesados en su camino derecho, y el otro de un total de n_2 nodos con l_2 nodos livianos y h_2 nodos pesados en su camino derecho. Es claro que el costo efectivo de mezclar estos dos montículos es de $l_1+h_1+l_2+h_2$, como evidencia el código precedente. Al realizar una mezcla, los únicos nodos que cambian su status de liviano/pesado son aquellos en los caminos derechos originales, ya que ningún otro nodo termina con sus subárboles alterados. También es claro que los nodos pesados en un camino derecho terminan siendo nodos livianos luego de la mezcla. En cuanto a los nodos livianos, pueden o no convertirse en pesados, pero como nuestro análisis es de peor caso, debemos asumir que eso sucede, de modo de tener el mayor cambio posible en el potencial. El cambio en el número de nodos livianos resulta entonces de $h_1 + h_2 - l_1 - l_2$, lo que restado del costo efectivo nos da $2(l_1 + l_2)$ como tiempo amortizado.

Resta mostrar que $l_1 + l_2 \in O(\log n)$. Como el tamaño del subárbol derecho de un nodo liviano es menos de la mitad del tamaño del árbol cuya raíz es dicho nodo, es evidente que el máximo número de nodos livianos en el camino derecho de un montículo de n nodos está en $O(\log n)$ en el peor caso (cuando todos los nodos en el camino derecho son livianos). Luego $l_1 + l_2 \in O(\log n_1 + \log n_2)$. Pero $\log n_1 + \log n_2 \in O(\log(n_1 + n_2))$, luego $l_1 + l_2 \in O(\log n)$. Ahora basta notar que, como `insert` y `deleteMin` no son sino `merge`s, sus costos efectivos y potenciales van a seguir el mismo patrón de razonamiento, y por lo tanto también tienen un costo amortizado de $O(\log n)$. Todo el análisis precedente es similar al realizado en un contexto imperativo, con la diferencia de que el potencial se mide allí por la cantidad de nodos pesados en vez de livianos, lo que no es ninguna sorpresa si recordamos que el método del físico en programas imperativos trata de acotar los ahorros y no las deudas. Por las mismas razones que nos llevaron al significado que tienen los nodos livianos en el camino derecho, es claro que el número de nodos pesados en dicho camino indica trabajo ahorrado al evitar el costo de inclinar la estructura del montículo hacia la izquierda.

5 Consideraciones adicionales

Como en un lenguaje funcional perezoso prácticamente todas las estructuras de datos diseñadas con cotas de peor caso como objetivo se demoran, se comparten y memoizan, es claro que se han transformado en estructuras de datos amortizadas, muchas veces innecesariamente. Se ha sugerido la conveniencia de disponer de un lenguaje de programación que permita ambos regímenes de evaluación, de modo de poder tener ambas clases de estructuras de datos, peor caso y amortizadas. En estas condiciones, es posible eliminar completamente la amortización, incorporando como parte de la estructura de datos un *cronograma*, que garantize el pago de deudas ejecutando prematuramente partes de cómputos demorados ([Oka96]). Si bien las técnicas de demostración de las cotas no se complican demasiado, la pérdida de claridad en el código es ostensible, indicando que esta transformación es una optimización avanzada y no una herramienta accesible al programador funcional común.

Por otra parte, al programar en un lenguaje funcional perezoso donde todas las estructuras convencionales se comportan con costos amortizados, la mayor claridad de las estructuras que son naturalmente amortizadas es un factor nada despreciable para un programador funcional. Además, el mayor overhead estructural comunmente asociado a las estructuras de peor caso pierde mucho de su sentido en este contexto, y comienza a jugar como un overhead innecesario que puede incluso hacerlas más lentas que las estructuras amortizadas correspondientes. Dado que para todas dichas estructuras de peor caso existen las correspondientes estructuras amortizadas, estas últimas resultan ser una herramienta muy importante para el programador funcional. Por ejemplo, se conocen formas funcionales de listas con concatenación ([Oka95b]), colas ([Oka95a]), y colas de dos extremos con concatenación ([Oka97]) en las que todas las operaciones toman tiempo amortizado constante.

En ausencia de un esquema útil para analizar en general programas funcionales bajo evaluación perezosa, las técnicas comentadas en el presente trabajo demuestran ser de gran importancia para recuperar el análisis de complejidad asintótica de los algoritmos. Como forma de concluír esta exploración del tema, conviene citar algunas sugerencias sobre el uso de amortización en programación funcional según se expresan en [Oka96]: utilizar la evaluación perezosa para hacer baratas a las futuras operaciones luego de la primera ejecución costosa, considerar la forma de acceso a la estructura de datos para decidir entre el método del banquero o el del físico, preferir procedimientos incrementales cuando sea posible, y recordar la existencia de optimizaciones que eliminan la amortización y restauran cotas de peor caso planificando la ejecución explícitamente.

Agradecimientos

El autor desea dar las gracias al Mgr. Pablo Fillottrani (Univ. Nac. del Sur, Bahía Blanca), quien lo introdujo en el tema del análisis amortizado durante el trabajo en común dentro la cátedra de él. Tambien agradece al Laboratório de Métodos Formais (PUC, Rio de Janeiro, Brasil) por suministrar las facilidades materiales usadas durante la redacción del presente trabajo.

6 Bibliografía

- [Has97] John Peterson, Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon Peyton Jones, Alastair Reid, Philip Wadler. *The Haskell Report, version 1.4. A Non-strict, Purely Functional Language.* 1997.

- [Hug89] John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2): 98-107, Abril 1989.

- [Kal91] Anne Kaldewaij y Berry Schoenmakers. The Derivation of a Tighter Bound for Top-down Skew Heaps. *Information Processing Letters*, 37, pp. 265-271, 1991.

- [Mor90] Bernard M.E. Moret y H.D. Shapiro. *Algorithms from P to NP: Design and Efficiency.* Volume 1. 1990, Benjamin/Cummings.

- [Oka95a] Chris Okasaki. Simple and Efficient Purely Functional Queues and Deques. *Journal of Functional Programming* 5(4), Octubre 1995, pp. 583-592.

- [Oka95b] Chris Okasaki. Amortization, Lazy Evaluation, and Persistence: Lists with Catenation via Lazy Linking. *Proceedings of the IEEE Symposium on Foundations of Computer Science*, Octubre 1995, pp. 646-654.

- [Oka96] Chris Okasaki. The Role of Lazy Evaluation in Amortized Data Structures. *Proceedings of the International Conference on Functional Programming*, Mayo 1996, pp. 62-72.

- [Oka97] Chris Okasaki. Catenable Double-Ended Queues. *Proceedings of the International Conference on Functional Programming*, 1997.

- [Sch97] Berry Schoenmakers. A Tight Lower Bound for Top-Down Skew Heaps. *Information Processing Letters*, 61(5), pp. 279-284, Marzo 1997.
- [Wei94] Mark A. Weiss. *Data Structures and Algorithm Analysis*. 2nd Edition. 1994, Benjamin/Cummings.