Purely Functional Data Structures

Advanced Data Structures
Final Work

Arnau Abella
Universitat Politècnica de Catalunya

June 2, 2020

1 Summary of the paper

The chosen paper, in fact a book, is *Purely Functional Data Structures* [Oka98].

The book has around 300 pages, 11 chapters and covers a wide variety of contents which, most of them, can't be given without previous content. For this reason, this work will focus only in chapter 5,6 and 7, which are, in my opinion, the most relevant introductory chapters. The list topics presented in these chapters are the following:

- Chapter 5. Fundamentals of Amortization
- Chapter 6. Amortization and Persistence via Lazy Evaluation
- Chapter 7. Eliminating Amortization

I'd love to talk about all different data structures mentioned in the book such as Binomial Heaps, Splay Heaps, Pairing Heaps, Bottom-Up Merge with Sharing, Hood-Melville Real-Time Queues, Binary Random-Access Lists, Skew Binomial Heaps, etc., and also about more advanced topics like *lazy rebuilding*, *numerical representations*, *data-strutural bootstrapping*, *implicit recursive slowdown*, etc. but, for today, we will work those 3 previously mentioned chapters and focus on a very simple and educative data structure, a *Queue*.

1.1 Fundamentals of Amortization

Given a sequence of operations, we may wish to know the running time of the entire sequence, but not care about the running time of any individual operation. Given a sequence of n operations, we wish to bound the total running time of the sequence by $O(n)$.

To prove an amortized bound, one defines the amortized cost of each operation and then proves that, for any sequence of operations, the total amortized cost of the operations is an upper on the total actual cost, i.e.,

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i \tag{1}$$

where a_i is the amortized cost of operation i, t_i is the actual cost of operation i, and m is the total number of operations.

The key to proving amortized bounds is to show that expensive operations occur only when the accumulated savings are sufficient to cover the remaining cost.

Tarjan [Tar85] describes two techniques for analyzing amortized data structures: the *banker's method* and the *physicist's method*.

In the *banker's method*, the accumulated savings are represented as *credits* that are associated with individual locations in the data structure. The amortized cost of any operation is defined to be the actual cost of the operation plus the credits allocated by the operation minus the credits spent by the operation, i.e.

$$a_i = t_i + c_i - \bar{c_i} \tag{2}$$

where c_i is the number of credits allocated by the operation i and $\bar{c_i}$ is the number of credits spent by operation i. Every credit must be allocated before it is spent, and no credit may be spent more than once. Therefore, $\sum c_i \geq \sum \bar{c_i}$, which in turn guarantees that $\sum a_i \geq \sum t_i$. Proofs using the banker's method typically define a *credit invariant* that regulates the distribution of credits in such a way that, whenever an expensive operation might occur, sufficient credits have been allocated in the right locations to cover its cost.

In the *physicist's method*, one describes a function Φ that maps each object d to a real number called the *potential* of d. The function Φ is tipically chosen

so that the potential is initially zero and is always non-negative. Then, the potential represents a lower bound on the accumulated savings.

Let d_i be the output of operation i and the input of operation $i+1$. Then, the amortized cost of operation i is defined to be the actual cost plus the change in potential between d_{i-1} and d_i, i.e.,

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1}) \tag{3}$$

The accumulated actual cost of the sequence of operations are

$$\begin{aligned}\sum_{i=1}^{j} t_i &= \sum_{i=1}^{j} (a_i + \Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \sum_{i=1}^{j} (\Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j)\end{aligned}$$

1.1.1 Queues

We next illustrate the banker's and pysicist's methods by analyzing a simple functional implementation of the FIFO queue abstraction (listing 1).

Both `snoc` and `head` run in $O(1)$ worst-case time, but `tail` takes $O(n)$ time in the worst-case. However, we can show that `snoc` and `tail` both take $O(1)$ amortized time using either the banker's method or the physicist's method.

Using the banker's method, we maintain a credit invariant that every element in the rear list is associated with a single credit. Every `snoc` into a non-empty queue takes one actual step, and allocates a credit to the new element of the rear list, for an amortized cost of two. Every `tail` that does not reverse the rear list takes one actual step and neither allocates nor spends any credits, for an amortized cost of one. Finally, every `tail` that does reverse the rear list takes $m+1$ actual steps, where m is the length of the rear list, and spends the m credits contained by that list, for an amortized cost of $m+1-m=1$.

Using the physicist's method, we define the potential function Φ to be the length of the rear list. Then every `snoc` into a non-empty queue takes one actual step and increases the potential by one, for an amortized cost of two.

```haskell
module Chapter5.BatchedQueue where

data Queue a = Queue [a] [a]

empty :: Queue a
empty = Queue [] []

checkf :: Queue a -> Queue a
checkf (Queue [] r) = Queue (reverse r) []
checkf q = q

isEmpty :: Queue a -> Bool
isEmpty (Queue f _) = null f

snoc :: a -> Queue a -> Queue a
snoc x (Queue f r) = checkf (Queue f (x:r))

head :: Queue a -> a
head (Queue [] _) = error "empty"
head (Queue (x:_) _) = x

tail :: Queue a -> Queue a
tail (Queue [] _) = error "empty"
tail (Queue (_:f) r) = checkf (Queue f r)
```

Listing 1: Functional Queue

Every `tail` that does not reverse the rear list takes one actual step and leaves the potential unchanged, for an amortized cost of one. Finally, every `tail` that does reverse the rear list takes $m + 1$ actual steps and sets the new rear list to `[]`, decreasing the potential by m, for an amortized cost of $m + 1 - m = 1$.

1.2 Amortization and Persistence via Lazy Evaluation

The amortized bounds break in the presence of persitence. In this chapter, we demonstrate how lazy evaluation can mediate the conflict between amortization and persistence, and adapt both the banker's and physicit's methods to account for lazy evaluation. We then illustrate the use of these new methods on a *Queue*.

1.2.1 Execution Traces and Logical Time

Traditional methods of amortization break in the presence of persistence because they assume a unique future, in which the accumulated savings will be spent at most once. Howeer, with persistence, multiple logical futures might all try to spend the same savings.

We model logical time with *execution traces*, which give an abstract view of the history of computation. An *execution trace* is a directed graph whose nodes represent operations of interest, usually just update operations on the data type in question. An edge from v to v' indicates that the operation v' uses some result of operation v. The *logical history* of operation v, denote $\hat{v}$, is the set of all operation on which the result of v depends (including v itself). In other words, $\hat{v}$ is the set of all nodes w such that there exists a path from w to v. A *logical future* of node v is any path from v to a terminal node. If there is more than one such path, then the node v has multiple logical futures.

Execution traces generalize the notion of *version graphs* [DSST89], which are often used to model the histories of persistent data structures.

1.2.2 Reconciling Amortization and Persistence

In this section, we show how the banker's and physicist's methods can be repaired by replacing the notion of accumulated savings with accumulated debt, where debt measure the cost of unevaluated lazy computations.

We must finde a way to guarantee that if the first application of f to x is expensive, then subsequent applications of f to x will not be. Without side-effects, this is impossible under *call-by-value* (i.e., strict evaluation) or *call-by-name* (i.e., lazy evaluation without memoization). Therefore, amortization cannot be usefully combined with persistence in languages supporting only these evaluation orders.

But now consider *call-by-need* (i.e., lazy evaluation with memoization). If x contains some suspended component that is needed by f, then the first

application of f to x forces the (potentially expensive) evaluation of that component and memoizes the result. Subsequent operations may then access the memoized result directly. This is exactly the desired behavior!

1.2.3 A Framework for Analyzing Lazy Data Structures

Historically, the most common technique for analyzing lazy programs has been to pretend that they are actually strict. We next describe a basic framework to support such analysis. In the remainder of this chapter, we adapt the banker's and physicist's methods to this framework, yielding both the first techniques for analysing persistent amortized data structures and the first practical techniques for analysing non-trivial lazy programs.

We classify the costs of any given operation into several categoriest. The *unshared cost* of an operation is the actual time it would take to execute the operation under the assumption that every suspension in the system at the beginning of the operation has already been forced and memoized. The *shared cost* of an operation is the time that it would take to execte every suspension created but not evaluated by the operation. The *complete cost* of an operation is the sum of its hared and unshared costs. Note that the complete cost is what the actual cost of the operation would be if lazy evaluation were replaced with strict evaluation.

Realized costs are the shared costs for suspensions that are executed during the overall computation. *Unrealized costs* are the shared costs for suspensions that are never executed. The *total actual cost* of a sequence of operations is the sum of the unshared costs and the realized shared costs.

We account for shared costs using the notion of *accumulated debt*. Initially, the debt is zero, but every time a suspensions is created, we increase the accumulated debt by the shared cost of the suspension (and any nested suspensions). Each operation then pays off a portion of the accumulated debt. The *amortized cost* of an operation is the unshared cost of the operation plus the amount of accumulated debt paid off by the operation. We are not allowed to force a suspensions until the debt associated with the suspension is entirely paid off.

We avoid the problem of reasoning about multiple logical futures by reasoning each logical future *as if it were the only one*. From the point of view of the operation that creates a suspension, any logical future that forces the suspension must itself pay for the suspension, any logical future that forces the suspension must itself pay for the suspensions. Using this method, we sometimes pays off a debt more than once, thereby overestimating the total

time required for a particular computation, but this does no harm and is a small price to pay for the simplicity of the resulting analyses.

1.2.4 The Banker's Method

We adapt the banker's method to account for accumulated debt rather than accumulated savings by replacing credits with debits. Each debit represents a constant amount of suspended work. When we initially suspend a given computation, we create a number of debits proportional to its shared cost and associate each debit with a location in the object. If the computation is *monolithic* (i.e., once begun, it runs to completition), then all debits are usually assigned to the root of the result. On the other hand, if the computation is *incremental* (i.e., decomposable into fragments that may be executed independently), then the debits may be distributed among the roots of the partial results.

The amortized cost of an operation is the unshared cost of the operation plus the number of debits discharged by the operation. Note that the number of debits created by an operation is *not* included in its amortized cost. To prove an amortized bound, we must show that, whenever we access a location (possibly triggering the executio nof a suspension), all debits associated with that location have already been discharged. Debits leftover at the end of the computation correspond to unrealized shared costs, and are irrelevant to the total actual cost.

Incremental functions play an important role in the banker's method because they allow debits to be dispersed to different locations in a data structure. Then, each location cab ne accessed as soon as its debits are discharged, without waiting for the debits at other locations to be discharged. In practice, this means that the initial partial result of an incremental computation can be paid for very quickly, and that subsequent partial results may be paid for as they are needed. Monolithic functions, on the other hand, are much less flexible.

The justification of this method is omitted for brevity as it does only brings correctness to this adaptation. Still, I would recommend reading the proof from *Pure Functional Data Structures* book.

1.2.5 Queues

1.3 Eliminating Amortization

2 The importance of Okasaki's work

Your personal evaluation on why the papers are relevant.

Okasaki works introduces a framework to remove amortized bounds using lazy evaluation which was unknown until then.

Okasaki also improve other techniques to analyze persistent data structures and present very simple but effective implementations of persistent data structures such as Real-Time Queues.

3 Experiment & Results

Design a set of experiments to validate the main aspects of the data structure.

A critical analysis of the results of the experiments (theoretical results).

4 Conclusion

Your personal conclusion.

References

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. K. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, February 1989.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, April 1985.