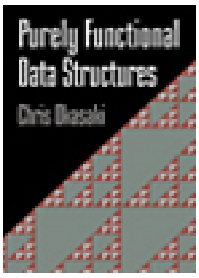

Purely Functional Data Structures

Chris Okasaki

Book DOI: http://dx.doi.org/10.1017/CBO9780511530104

Online ISBN: 9780511530104

Hardback ISBN: 9780521631242

Paperback ISBN: 9780521663502

Chapter

6 - Amortization and Persistence via Lazy Evaluation pp. 57-82

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.007

Cambridge University Press

6
Amortization and Persistence via Lazy Evaluation

The previous chapter introduced the idea of amortization and gave several examples of data structures with good amortized bounds. However, for each these data structures, the amortized bounds break in the presence of persistence. In this chapter, we demonstrate how lazy evaluation can mediate the conflict between amortization and persistence, and adapt both the banker's and physicist's methods to account for lazy evaluation. We then illustrate the use of these new methods on several amortized data structures that use lazy evaluation internally.

6.1 Execution Traces and Logical Time

In the previous chapter, we saw that 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. However, with persistence, multiple logical futures might all try to spend the same savings. But what exactly do we mean by the "logical future" of an operation?

We model logical time with *execution traces*, which give an abstract view of the history of a 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 operation v' uses some result of operation v. The *logical history* of operation v, denoted $\hat{v}$, is the set of all operations 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 (possibly of length 0) from w to v. A *logical future* of a node v is any path from v to a terminal node (i.e., a node with out-degree zero). If there is more than one such path, then node v has multiple logical futures. We sometimes refer to the logical history or logical future of an object, meaning the logical history or logical future of the operation that created the object.

Exercise 6.1 Draw an execution trace for the following set of operations. Annotate each node in the trace with the number of logical futures at that node.

```
val a = snoc (empty, 0)
val b = snoc (a, 1)
val c = tail b
val d = snoc (b, 2)
val e = c ++ d
val f = tail c
val g = snoc (d, 3)
```

◇

Execution traces generalize the notion of *version graphs* [DSST89], which are often used to model the histories of persistent data structures. In a version graph, nodes represent the various versions of a single persistent identity and edges represent dependencies between versions. Thus, version graphs model the results of operations and execution traces model the operations themselves. Execution traces are often more convenient for combining the histories of several persistent identities (perhaps not even of the same type) or for reasoning about operations that do not return a new version (e.g., queries) or that return several results (e.g., splitting a list into two sublists).

For ephemeral data structures, the out-degree of every node in a version graph or execution trace is typically restricted to be at most one, reflecting the limitation that objects can be updated at most once. To model various flavors of persistence, version graphs allow the out-degree of every node to be unbounded, but make other restrictions. For instance, version graphs are often limited to be trees (forests) by restricting the in-degree of every node to be at most one. Other version graphs allow in-degrees of greater than one, but forbid cycles, making every graph a dag. We make none of these restrictions on execution traces for persistent data structures. Nodes with in-degree greater than one correspond to operations that take more than one argument, such as list catenation or set union. Cycles arise from recursively defined objects, which are supported by many lazy languages. We even allow multiple edges between a single pair of nodes, as might occur if a list is catenated with itself.

We will use execution traces in Section 6.3.1 when we extend the banker's method to cope with persistence.

6.2 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 measures the cost of unevaluated lazy computations. The intuition is that, although savings can only be spent once, it does no harm to pay off debt more than once.

6.2.1 The Role of Lazy Evaluation

Recall that an *expensive* operation is one whose actual costs are greater than its (desired) amortized costs. For example, suppose some application $f\ x$ is expensive. With persistence, a malicious adversary might call $f\ x$ arbitrarily often. (Note that each operation is a new logical future of x.) If each operation takes the same amount of time, then the amortized bounds degrade to the worst-case bounds. Hence, we must find 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), because every application of f to x takes exactly the same amount of time. 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!

Remark In retrospect, the relationship between lazy evaluation and amortization is not surprising. Lazy evaluation can be viewed as a form of self-modification, and amortization often involves self-modification [ST85, ST86b]. However, lazy evaluation is a particularly disciplined form of self-modification — not all forms of self-modification typically used in amortized ephemeral data structures can be encoded as lazy evaluation. In particular, splay trees do not appear to be amenable to this technique.

6.2.2 A Framework for Analyzing Lazy Data Structures

We have just shown that lazy evaluation is necessary to implement amortized data structures purely functionally. Unfortunately, analyzing the running times of programs involving lazy evaluation is notoriously difficult. Historically, the most common technique for analyzing lazy programs has been to pretend that they are actually strict. However, this technique is completely inadequate for analyzing lazy amortized data structures. We next describe a basic framework to support such analyses. In the remainder of this chapter, we adapt the banker's and physicist's methods to this framework, yielding both the first techniques for analyzing persistent amortized data structures and the first practical techniques for analyzing non-trivial lazy programs.

We classify the costs of any given operation into several categories. First, 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 (i.e., under the assumption that force always takes $O(1)$ time, except for those suspensions that are created and forced within the same operation). The *shared cost* of an operation is the time that it would take to execute every suspension created but not evaluated by the operation (under the same assumption as above). The *complete cost* of an operation is the sum of its shared 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.

We further partition the total shared costs of a sequence of operations into realized and unrealized costs. *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—unrealized costs do not contribute to the actual cost. Note that the amount that any particular operation contributes to the total actual cost is at least its unshared cost, and at most its complete cost, depending on how much of its shared cost is realized.

We account for shared costs using the notion of *accumulated debt*. Initially, the accumulated debt is zero, but every time a suspension 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 suspension until the debt associated with the suspension is entirely paid off.

Remark An amortized analysis based on the notion of accumulated debt works a lot like a *layaway plan*. In a layaway plan, you find something—a diamond ring, say—that you want to buy, but that you can't afford to pay for yet. You agree on a price with the jewelry store and ask them to set the ring aside in your name. You then make regular payments, and receive the ring only when it is entirely paid off.

In analyzing a lazy data structure, you find a computation that you can't afford to execute yet. You create a suspension for the computation, and assign the suspension an amount of debt proportional to its shared cost. You then pay off the debt a little at a time. Finally, when the debt is entirely paid off, you are allowed to execute the suspension. ◇

There are three important moments in the life cycle of a suspension: when it is created, when it is entirely paid off, and when it is executed. The proof obligation is to show that the second moment precedes the third. If every suspension is paid off before it is forced, then the total amount of debt that has been paid off is an upper bound on the realized shared costs, and therefore the total amortized cost (i.e., the total unshared cost plus the total amount of debt that has been paid off) is an upper bound on the total actual cost (i.e., the total unshared cost plus the realized shared costs). We will formalize this argument in Section 6.3.1.

One of the most difficult problems in analyzing the running time of lazy programs is reasoning about the interactions of multiple logical futures. We avoid this problem by reasoning about 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. If two logical futures wish to force the same suspension, then both must pay for the suspension individually—they may not cooperate and each pay only a portion of the debt. An alternative view of this restriction is that we are allowed to force a suspension *only when the debt for that suspension has been paid off within the logical history of the current operation*. Using this method, we sometimes pay 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.

6.3 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. The choice of location for each debit depends on the nature of the computation. If the computation is *monolithic* (i.e., once begun, it runs to completion), 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. The order in which debits should be discharged depends on how the object will be accessed; debits on nodes likely to be accessed soon should be discharged first. To prove an amortized bound, we must show that, whenever we access a location (possibly triggering the execution of a suspension), all debits associated with that location have already been discharged (and hence the suspended computation has been entirely paid off). This guarantees that the total number of debits discharged by a sequence of operations is an upper bound on the realized shared costs of the operations. The total amortized costs are therefore an upper bound on the total actual costs. Debits leftover at the end of the computation correspond to unrealized shared costs, and are irrelevant to the total actual costs.

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, each corresponding to a nested suspension. Then, each location can be 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 results 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 programmer must anticipate when the result of an expensive monolithic computation will be needed, and set up the computation far enough in advance to be able to discharge all its debits by the time its result is needed.

6.3.1 Justifying the Banker's Method

In this section, we justify the claim that the total amortized cost is an upper bound on the total actual cost. The total amortized cost is the total unshared cost plus the total number of debits discharged (counting duplicates); the total actual cost is the total unshared cost plus the realized shared costs. Therefore, we must show that the total number of debits discharged is an upper bound on the realized shared costs.

We can view the banker's method abstractly as a graph labelling problem, using the execution traces of Section 6.1. The problem is to label every node in a trace with three (multi)sets, $s(v)$, $a(v)$, and $r(v)$, such that

$$\begin{array}{ll} \text{(I)} & v \neq v' \Rightarrow s(v) \cap s(v') = \emptyset \\ \text{(II)} & a(v) \subseteq \bigcup_{w \in \hat{v}} s(w) \\ \text{(III)} & r(v) \subseteq \bigcup_{w \in \hat{v}} a(w) \end{array}$$

$s(v)$ is a set, but $a(v)$ and $r(v)$ may be multisets (i.e., may contain duplicates). Conditions II and III ignore duplicates.

$s(v)$ is the set of debits allocated by operation v. Condition I states that no debit may be allocated more than once. $a(v)$ is the multiset of debits discharged by v. Condition II insists that no debit may be discharged before it is created, or more specifically, that an operation can only discharge debits that appear in its logical history. Finally, $r(v)$ is the multiset of debits *realized* by v—that is, the multiset of debits corresponding to the suspensions forced by v. Condition III requires that no debit may be realized before it is discharged, or more specifically, that no debit may realized unless it has been discharged within the logical history of the current operation.

Why are $a(v)$ and $r(v)$ multisets rather than sets? Because a single operation might discharge the same debits more than once or realize the same debits more than once (by forcing the same suspensions more than once). Although we never deliberately discharge the same debit more than once, it could happen if we were to combine a single object with itself. For example, suppose in some analysis of a list catenation function, we discharge a few debits from the first argument and a few debits from the second argument. If we then catenate a list with itself, we might discharge the same few debits twice.

Given this abstract view of the banker's method, we can easily measure various costs of a computation. Let V be the set of all nodes in the execution trace. Then, the total shared cost is $\sum_{v \in V} |s(v)|$ and the total number of debits discharged is $\sum_{v \in V} |a(v)|$. Because of memoization, the realized shared cost is not $\sum_{v \in V} |r(v)|$, but rather $|\bigcup_{v \in V} r(v)|$, where $\bigcup$ discards duplicates. Thus, a suspension that is forced multiple times contributes only once to the actual cost. By Condition III, we know that $\bigcup_{v \in V} r(v) \subseteq \bigcup_{v \in V} a(v)$. Therefore,

$$|\bigcup_{v \in V} r(v)| \leq |\bigcup_{v \in V} a(v)| \leq \sum_{v \in V} |a(v)|$$

So the realized shared cost is bounded by the total number of debits discharged, and the total actual cost is bounded by the total amortized cost, as desired.

Remark This argument once again emphasizes the importance of memoization. Without memoization (i.e., if we were using call-by-name rather than call-by-need), the total realized cost would be $\sum_{v \in V} |r(v)|$, and there is no reason to expect this sum to be less than $\sum_{v \in V} |a(v)|$.

6.3.2 Example: Queues

We next develop an efficient persistent implementation of queues, and prove that every operation runs in $O(1)$ amortized time using the banker's method.

Based on the discussion in the previous section, we must somehow incorporate lazy evaluation into the design of the data structure, so we replace the pair of lists in the simple queues of Section 5.2 with a pair of streams.† To simplify later operations, we also explicitly track the lengths of the two streams.

```
type α Queue = int × α Stream × int × α Stream
```

The first integer is the length of the front stream and the second integer is the length of the rear stream. Note that a pleasant side effect of maintaining this length information is that we can trivially support a constant-time size function.

Now, waiting until the front list becomes empty to reverse the rear list does not leave sufficient time to pay for the reverse. Instead, we periodically *rotate* the queue by moving all the elements of the rear stream to the end of the front stream, replacing f with f ++ reverse r and setting the new rear stream to empty. Note that this transformation does not affect the relative ordering of the elements.

When should we rotate the queue? Recall that reverse is a monolithic function. We must therefore set up the computation far enough in advance to be able to discharge all its debits by the time its result is needed. The reverse computation takes $|r|$ steps, so we allocate $|r|$ debits to account for its cost. (For now we ignore the cost of the ++ operation). The earliest the reverse suspension could be forced is after $|f|$ applications of tail, so if we rotate the queue when $|r| \approx |f|$ and discharge one debit per operation, then we will have paid for the reverse by the time it is executed. In fact, we rotate the queue whenever r becomes one longer than f, thereby maintaining the invariant that $|f| \geq |r|$. Incidentally, this guarantees that f is empty only if r is also empty, as in the simple queues of Section 5.2. The major queue functions can now be written as follows:

```
fun snoc ((lenf, f, lenr, r) , x) = check (lenf, f, lenr+1, $CONS (x, r))
fun head (lenf, $CONS (x, f'), lenr, r) = x
fun tail (lenf, $CONS (x, f'), lenr, r) = check (lenf−1, f', lenr, r)
```

where the helper function check guarantees that $|f| \geq |r|$.

```
fun check (q as (lenf, f, lenr, r)) =
      if lenr ≤ lenf then q else (lenf+lenr, f ++ reverse r, 0, $NIL)
```

† Actually, it would be enough to replace only the front list with a stream, but we replace both for simplicity.

```
structure BankersQueue : QUEUE =
struct
  type α Queue = int × α Stream × int × α Stream

  val empty = (0, $NIL, 0, $NIL)
  fun isEmpty (lenf, _, _, _) = (lenf = 0)

  fun check (q as (lenf, f, lenr, r)) =
        if lenr ≤ lenf then q else (lenf+lenr, f ++ reverse r, 0, $NIL)

  fun snoc ((lenf, f, lenr, r), x) = check (lenf, f, lenr+1, $CONS (x, r))

  fun head (lenf, $NIL, lenr, r) = raise EMPTY
    | head (lenf, $CONS (x, f'), lenr, r) = x
  fun tail (lenf, $NIL, lenr, r) = raise EMPTY
    | tail (lenf, $CONS (x, f'), lenr, r) = check (lenf−1, f', lenr, r)
end
```

Figure 6.1. Amortized queues using the banker's method.

The complete code for this implementation appears in Figure 6.1.

To understand how this implementation deals efficiently with persistence, consider the following scenario. Let q_0 be some queue whose front and rear streams are both of length m, and let q_i = tail q_{i-1}, for $0 < i \leq m + 1$. The queue is rotated during the first application of tail, and the reverse suspension created by the rotation is forced during the last application of tail. This reversal takes m steps, and its cost is amortized over the sequence $q_1 \ldots q_m$. (For now, we are concerned only with the cost of the reverse—we ignore the cost of the ++.)

Now, choose some branch point k, and repeat the calculation from q_k to q_{m+1}. (Note that q_k is used persistently.) Do this d times. How often is the reverse executed? It depends on whether the branch point k is before or after the rotation. Suppose k is after the rotation. In fact, suppose $k = m$ so that each of the repeated branches is a single tail. Each of these branches forces the reverse suspension, but they each force the *same* suspension, so the reverse is executed only once. Memoization is crucial here—without memoization, the reverse would be re-executed each time, for a total cost of $m(d+1)$ steps, with only $m + 1 + d$ operations over which to amortize this cost. For large d, this would result in an $O(m)$ amortized cost per operation, but memoization gives us an amortized cost of only $O(1)$ per operation.

It is possible to re-execute the reverse however. Simply take $k = 0$ (i.e., make the branch point just before the rotation). Then the first tail of each branch repeats the rotation and creates a new reverse suspension. This new suspension is forced in the last tail of each branch, executing the reverse. Because these are different suspensions, memoization does not help at all. The total cost of all the reversals is $m \cdot d$, but now we have $(m+1)(d+1)$ operations over which to amortize this cost, again yielding an amortized cost of $O(1)$ per operation. The key is that we duplicate work only when we also duplicate the sequence of operations over which to amortize the cost of that work.

This informal argument shows that these queues require only $O(1)$ amortized time per operation even when used persistently. We formalize this proof using the banker's method.

By inspection, the unshared cost of every queue operation is $O(1)$. Therefore, to show that the amortized cost of every queue operation is $O(1)$, we must prove that discharging $O(1)$ debits per operation suffices to pay off every suspension before it is forced. In fact, only snoc and tail discharge any debits.

Let $d(i)$ be the number of debits on the ith node of the front stream and let $D(i) = \sum_{j=0}^{i} d(j)$ be the cumulative number of debits on all nodes up to and including the ith node. We maintain the following *debit invariant*:

$$D(i) \leq \min(2i, |f| - |r|)$$

The $2i$ term guarantees that all debits on the first node of the front stream have been discharged (since $d(0) = D(0) \leq 2 \cdot 0 = 0$), so this node may be forced at will (for instance, by head or tail). The $|f| - |r|$ term guarantees that all debits in the entire queue have been discharged whenever the streams are of equal length, which happens just before the next rotation.

Theorem 6.1 snoc *and* tail *maintain the debit invariant by discharging one and two debits, respectively.*

Proof Every snoc that does not cause a rotation simply adds a new element to the rear stream, increasing $|r|$ by one and decreasing $|f| - |r|$ by one. This violates the invariant at any node for which $D(i)$ was previously equal to $|f| - |r|$. We can restore the invariant by discharging the first debit in the queue, which decreases every subsequent cumulative debit total by one. Similarly, every tail that does not cause a rotation simply removes an element from the front stream. This decreases $|f|$ by one (and hence $|f| - |r|$ by one), but, more importantly, it decreases the index i of every remaining node by one, which in turn decreases $2i$ by two. Discharging the first two debits in the queue restores the invariant. Finally, consider a snoc or tail that causes a rotation. Just before the rotation, we are guaranteed that all debits in the queue have been discharged, so, after the rotation, the only undischarged debits are those generated by the rotation itself. If $|f| = m$ and $|r| = m + 1$ at the time of the rotation, then we create m debits for the append and $m + 1$ debits for the reverse. The append function is incremental so we place one of its debits on each of the first m nodes. On the other hand, the reverse function is monolithic so we place all $m + 1$ of its debits on node m, the first node of the reversed stream. Thus, the debits are distributed such that

$$d(i) = \begin{cases} 1 & \text{if } i < m \\ m+1 & \text{if } i = m \\ 0 & \text{if } i > m \end{cases} \quad \text{and} \quad D(i) = \begin{cases} i+1 & \text{if } i < m \\ 2m+1 & \text{if } i \geq m \end{cases}$$

This distribution violates the invariant at both node 0 and node m, but discharging the debit on node 0 restores the invariant at both locations. □

The format of this argument is typical. Debits are distributed across several nodes for incremental functions, and all on the same node for monolithic functions. Debit invariants measure, not just the number of debits on a given node, but the number of debits along the path from the root to the given node. This reflects the fact that accessing a node requires first accessing all its ancestors. Therefore, the debits on all those nodes must be zero as well.

This data structure also illustrates a subtle point about nested suspensions—the debits for a nested suspension may be allocated, and even discharged, before the suspension is physically created. For example, consider how ++ works. The suspension for the second node in the stream is not physically created until the suspension for the first node is forced. However, because of memoization, the suspension for the second node will be shared whenever the suspension for the first node is shared. Therefore, we consider a nested suspension to be implicitly created at the time that its enclosing suspension is created. Furthermore, when considering debit arguments or otherwise reasoning about the shape of an object, we ignore whether a node has been physically created or not. Rather we reason about the shape of an object as if all nodes were in their final form, i.e., as if all suspensions in the object had been forced.

Exercise 6.2 Suppose we change the invariant from $|f| \geq |r|$ to $2|f| \geq |r|$.

(a) Prove that the $O(1)$ amortized bounds still hold.
(b) Compare the relative performance of the two implementations on a sequence of one hundred snocs followed by one hundred tails.

6.3.3 Debit Inheritance

We frequently create suspensions whose bodies force other, existing suspensions. We say that the new suspension *depends* on the older suspensions. In the queue example, the suspension created by reverse *r* depends on *r*, and the suspension created by f ++ reverse r depends on f. Whenever we force a suspension, we must be sure that we have discharged not only all the debits for that suspension, but also all the debits for any suspensions on which it depends. In the queue example, the debit invariant guarantees that we create new suspensions using ++ and reverse only when the existing suspensions have been entirely paid off. However, we will not always be so lucky.

When we create a suspension that depends on an existing suspension with undischarged debits, we reassign those debits to the new suspension. We say that the new suspension *inherits* the debits of the older suspension. We may not force the new suspension until we have discharged both the new suspension's own debits and the debits it inherited from the older suspension. The banker's method makes no distinction between the two sets of debits, treating them all as if they belong to the new suspension. We will use debit inheritance to analyze data structures in Chapters 9, 10, and 11.

Remark Debit inheritance assumes that there is no way to access the older suspension in the current object other than through the new suspension. For example, debit inheritance could not be used in analyzing the following function on pairs of streams:

```
fun reverseSnd (xs, ys) = (reverse ys, ys)
```

Here, we can force *ys* through either the first component of the pair or the second component of the pair. In such situations, we either duplicate the debits on *ys* and let the new suspension inherit the duplicates, or keep one copy of each debit and explicitly track the dependencies.

6.4 The Physicist's Method

Like the banker's method, the physicist's method can also be adapted to work with accumulated debt rather than accumulated savings. In the traditional physicist's method, one describes a potential function Φ that represents a lower bound on the accumulated savings. To work with debt instead of savings, we replace Φ with a function Ψ that maps each object to a potential representing an upper bound on the accumulated debt (or at least, an upper bound on this object's portion of the accumulated debt). Roughly speaking, the amortized cost of an operation is then the complete cost of the operation (i.e., the shared and unshared costs) minus the change in potential. Recall that an easy way to calculate the complete cost of an operation is to pretend that all computation is strict.

Any changes in the accumulated debt are reflected by changes in the potential. If an operation does not pay any shared costs, then the change in potential is equal to its shared cost, so the amortized cost of the operation is equal to its unshared cost. On the other hand if an operation does pay some of its shared cost, or shared costs of previous operations, then the change in potential is smaller than the shared cost (i.e., the accumulated debt increases by less than the shared cost), so the amortized cost of the operation is greater than its unshared cost. However, the amortized cost of an operation can never be less than its unshared cost, so the change in potential is not allowed to be more than the shared cost.

We can justify the physicist's method by relating it back to the banker's method. Recall that in the banker's method, the amortized cost of an operation was its unshared cost plus the number of debits discharged. In the physicist's method, the amortized cost is the complete cost minus the change in potential, or, in other words, the unshared cost plus the difference between the shared cost and the change in potential. If we consider one unit of potential to be equivalent to one debit, then the shared cost is the number of debits by which the accumulated debt could have increased, and the change in potential is the number of debits by which the accumulated debt did increase. The difference must have been made up by discharging some debits. Therefore, the amortized cost in the physicist's method can also be viewed as the unshared cost plus the number of debits discharged.

Sometimes, we wish to force a suspension in an object when the potential of the object is not zero. In that case, we add the object's potential to the amortized cost. This typically happens in queries, where the cost of forcing the suspension cannot be reflected by a change in potential because the operation does not return a new object.

The major difference between the banker's and physicist's methods is that, in the banker's method, we are allowed to force a suspension as soon as the debits for that suspension have been paid off, without waiting for the debits for other suspensions to be discharged, but in the physicist's method, we can force a shared suspension only when we have reduced the entire accumulated debt of an object, as measured by the potential, to zero. Since potential measures only the accumulated debt of an object as a whole and does not distinguish between different locations, we must pessimistically assume that the entire outstanding debt is associated with the particular suspension we wish to force. For this reason, the physicist's method appears to be less powerful than the banker's method. However, when it applies, the physicist's method tends to be much simpler than the banker's method.

Since the physicist's method cannot take advantage of the piecemeal execution of nested suspensions, there is no reason to prefer incremental functions over monolithic functions. In fact, a good hint that the physicist's method might be applicable is if all or most suspensions are monolithic.

6.4.1 Example: Binomial Heaps

In Chapter 5, we showed that the binomial heaps of Section 3.2 support insert in $O(1)$ amortized time. However, this bound degrades to $O(\log n)$ worst-case time if the heaps are used persistently. With lazy evaluation, we can restore the $O(1)$ amortized time bound such that it holds regardless of whether the heaps are used persistently.

The key is to change the representation of heaps from a list of trees to a suspended list of trees.

```
type Heap = Tree list susp
```

Then we can rewrite insert as

```
fun lazy insert (x, $ts) = $insTree (NODE (0, x, [ ]), ts)
```

or, equivalently, as

```
fun insert (x, h) = $insTree (NODE (0, x, [ ]), force h)
```

The remaining functions are equally easy to rewrite, and are shown in Figure 6.2.

Next, we analyze the amortized running time of insert. Since insert is monolithic, we use the physicist's method. First, we define the potential function to be $\Psi(h) = Z(|h|)$, where $Z(n)$ is the number of zeros in the (minimum length) binary representation of n. Next, we show that the amortized cost of inserting an element into a binomial heap of size n is two. Suppose that the lowest k digits in the binary representation of n are ones. Then the complete cost of insert is proportional to $k + 1$, eventually including k calls to link. Now, consider the change in potential. The lowest k digits change from ones to zeros and the next digit changes from zero to one, so the change in potential is $k - 1$. The amortized cost is therefore $(k + 1) - (k - 1) = 2$.

Remark Note that this proof is dual to the one given in Section 5.3. There the potential was the number of ones in the binary representation of n; here it is the number of zeros. This reflects the dual nature of accumulated savings and accumulated debt.

Exercise 6.3 Prove that findMin, deleteMin, and merge also run in $O(\log n)$ amortized time.

```
functor LazyBinomialHeap (Element : ORDERED) : HEAP =
struct
   structure Elem = Element

   datatype Tree = NODE of int × Elem.T × Tree list
   type Heap = Tree list susp

   val empty = $[ ]
   fun isEmpty ($ts) = null ts

   fun rank (NODE (r, x, c)) = r
   fun root (NODE (r, x, c)) = x
   fun link (t₁ as NODE (r, x₁, c₁), t₂ as NODE (_, x₂, c₂)) =
            if Elem.leq (x₁, x₂) then NODE (r+1, x₁, t₂ :: c₁)
            else NODE (r+1, x₂, t₁ :: c₂)
   fun insTree (t, [ ]) = [t]
      | insTree (t, ts as t' :: ts') =
            if rank t < rank t' then t :: ts else insTree (link (t, t'), ts')

   fun mrg (ts₁, [ ]) = ts₁
      | mrg ([ ], ts₂) = ts₂
      | mrg (ts₁ as t₁ :: ts₁', ts₂ as t₂ :: ts₂') =
            if rank t₁ < rank t₂ then t₁ :: mrg (ts₁', ts₂)
            else if rank t₂ < rank t₁ then t₂ :: mrg (ts₁, ts₂')
            else insTree (link (t₁, t₂), mrg (ts₁', ts₂'))

   fun lazy insert (x, $ts) = $insTree (NODE (0, x, [ ]), ts)
   fun lazy merge ($ts₁, $ts₂) = $mrg (ts₁, ts₂)

   fun removeMinTree [ ] = raise EMPTY
      | removeMinTree [t] = (t, [ ])
      | removeMinTree (t :: ts) =
            let val (t', ts') = removeMinTree ts
            in if Elem.leq (root t, root t') then (t, ts) else (t', t :: ts') end

   fun findMin ($ts) = let val (t, _) = removeMinTree ts in root t end
   fun lazy deleteMin ($ts) =
            let val (NODE (_, x, ts₁), ts₂) = removeMinTree ts
            in $mrg (rev ts₁, ts₂) end
end
```

Figure 6.2. Lazy binomial heaps.

Exercise 6.4 Suppose that we remove the **lazy** keyword from the definitions of merge and deleteMin, so that these functions evaluate their arguments immediately. Show that both functions still run in $O(\log n)$ amortized time.

Exercise 6.5 An unfortunate consequence of suspending the list of trees is that the running time of isEmpty degrades from $O(1)$ worst-case time to $O(\log n)$ amortized time. Restore the $O(1)$ running time of isEmpty by explicitly maintaining the size of every heap. Rather than modifying this implementation directly, implement a functor SizedHeap, similar to the ExplicitMin functor of Exercise 3.7, that transforms any implementation of heaps into one that explicitly maintains the size.

6.4.2 Example: Queues

We next adapt our implementation of queues to use the physicist's method. Again, we show that every operation takes only $O(1)$ amortized time.

Because there is no longer any reason to prefer incremental suspensions over monolithic suspensions, we use suspended lists instead of streams. In fact, the rear list need not be suspended at all, so we represent it with an ordinary list. Again, we explicitly track the lengths of the lists and guarantee that the front list is always at least as long as the rear list.

Since the front list is suspended, we cannot access its first element without executing the entire suspension. We therefore keep a working copy of a prefix of the front list to answer head queries. This working copy is represented as an ordinary list for efficient access, and is non-empty whenever the front list is non-empty. The final type is

```
type α Queue = α list × int × α list susp × int × α list
```

The major functions on queues may then be written

```
fun snoc ((w, lenf, f, lenr, r), x) = check (w, lenf, f, lenr+1, x :: r)
fun head (x :: w, lenf, f, lenr, r) = x
fun tail (x :: w, lenf, f, lenr, r) = check (w, lenf−1, $tl (force f), lenr, r)
```

The helper function check enforces two invariants: that *r* is no longer than *f*, and that *w* is non-empty whenever *f* is non-empty.

```
fun checkw ([ ], lenf, f, lenr, r) = (force f, lenf, f, lenr, r)
  | checkw q = q
fun check (q as (w, lenf, f, lenr, r)) =
      if lenr ≤ lenf then checkw q
      else let val f′ = force f
           in checkw (f′, lenf+lenr, $(f′ @ rev r), 0, [ ]) end
```

The complete implementation of these queues appears in Figure 6.3.

To analyze these queues using the physicist's method, we choose a potential function Ψ in such a way that the potential is zero whenever we force the suspended list. This happens in two situations: when *w* becomes empty and when *r* becomes longer than *f*. We therefore choose Ψ to be

$$\Psi(q) = \min(2|w|, |f| - |r|)$$

```
structure PhysicistsQueue : QUEUE =
struct
  type α Queue = α list × int × α list susp × int × α list

  val empty = ([ ], 0, $[ ], 0, [ ])
  fun isEmpty (_, lenf, _, _, _) = (lenf = 0)

  fun checkw ([ ], lenf, f, lenr, r) = (force f, lenf, f, lenr, r)
    | checkw q = q
  fun check (q as (w, lenf, f, lenr, r)) =
        if lenr ≤ lenf then checkw q
        else let val f' = force f
             in checkw (f', lenf+lenr, $(f' @ rev r), 0, [ ]) end

  fun snoc ((w, lenf, f, lenr, r), x) = check (w, lenf, f, lenr+1, x :: r)

  fun head ([ ], lenf, f, lenr, r) = raise EMPTY
    | head (x :: w, lenf, f, lenr, r) = x
  fun tail ([ ], lenf, f, lenr, r) = raise EMPTY
    | tail (x :: w, lenf, f, lenr, r) = check (w, lenf−1, $tl (force f), lenr, r)
end
```

Figure 6.3. Amortized queues using the physicist's method.

Theorem 6.2 *The amortized costs of* snoc *and* tail *are at most two and four, respectively.*

Proof Every snoc that does not cause a rotation simply adds a new element to the rear list, increasing $|r|$ by one and decreasing $|f| - |r|$ by one. The complete cost of the snoc is one, and the decrease in potential is at most one, for an amortized cost of at most $1 - (-1) = 2$. Every tail that does not cause a rotation removes the first element from the working list and lazily removes the same element from the front list. This decreases $|w|$ by one and $|f| - |r|$ by one, which decreases the potential by at most two. The complete cost of tail is two, one for the unshared costs (including removing the first element from w) and one for the shared cost of lazily removing the head of f. The amortized cost is therefore at most $2 - (-2) = 4$.

Finally, consider a snoc or tail that causes a rotation. In the initial queue, $|f| = |r|$, so $\Psi = 0$. Just before the rotation, $|f| = m$ and $|r| = m + 1$. The shared cost of the rotation is $2m + 1$ and the potential of the resulting queue is $2m$. The amortized cost of snoc is thus $1 + (2m + 1) - 2m = 2$. The amortized cost of tail is $2 + (2m + 1) - 2m = 3$. (The difference is that tail must also account for the shared cost of removing the first element of f.) □

```
signature SORTABLE =
sig
  structure Elem : ORDERED

  type Sortable

  val empty : Sortable
  val add   : Elem.T × Sortable → Sortable
  val sort  : Sortable → Elem.T list
end
```

Figure 6.4. Signature for sortable collections.

Exercise 6.6 Show why each of the following proposed "optimizations" actually breaks the $O(1)$ amortized time bounds. These examples illustrate common mistakes in designing persistent amortized data structures.

(a) Observe that check forces *f* during a rotation and installs the result in *w*. Wouldn't it be lazier, and therefore better, to never force *f* until *w* becomes empty?

(b) Observe that, during a tail, we replace *f* with $tl (force *f*). Creating and forcing suspensions have non-trivial overheads that, even if $O(1)$, can contribute to a large constant factor. Wouldn't it be lazier, and therefore better, to not change *f*, but instead to merely decrement *lenf* to indicate that the element has been removed? ◇

6.4.3 Example: Bottom-Up Mergesort with Sharing

The majority of examples in the remaining chapters use the banker's method rather than the physicist's method. Therefore, we give one more example of the physicist's method here.

Imagine that you want to sort several similar lists, such as *xs* and *x* :: *xs*, or *xs* @ *zs* and *ys* @ *zs*. For efficiency, you wish to take advantage of the fact that these lists share common tails, so that you do not repeat the work of sorting those tails. We call an abstract data type for this problem a *sortable collection*. A signature for sortable collections is given in Figure 6.4.

Now, if we create a sortable collection *xs'* by adding each of the elements in *xs*, then we can sort both *xs* and *x* :: *xs* by calling sort *xs'* and sort (add (*x*, *xs'*)).

We could implement sortable collections as balanced binary search trees. Then add and sort would run in $O(\log n)$ worst-case time and $O(n)$ worst-case time, respectively. We achieve the same bounds, but in an amortized sense, using *bottom-up mergesort*.

Bottom-up mergesort first splits a list into n ordered segments, where each segment initially contains a single element. It then merges equal-sized segments in pairs until only one segment of each size remains. Finally, segments of unequal size are merged, from smallest to largest.

Suppose we take a snapshot just before the final cleanup phase. Then the sizes of all segments are distinct powers of 2, corresponding to the one bits of n. This is the representation we will use for sortable collections. Then similar collections share all the work of bottom-up mergesort except for the final cleanup phase merging unequal-sized segments. The complete representation is a suspended list of segments, each of which is list of elements, together with an integer representing the total size of the collection.

```
type Sortable = int × Elem.T list list susp
```

The individual segments are stored in increasing order of size, and the elements in each segment are stored in increasing order as determined by the comparison functions in the Elem structure.

The fundamental operation on segments is mrg, which merges two ordered lists.

```
fun mrg ([ ], ys) = ys
  | mrg (xs, [ ]) = xs
  | mrg (xs as x :: xs', ys as y :: ys') =
      if Elem.leq (x, y) then x :: mrg (xs', ys) else y :: mrg (xs, ys')
```

To add a new element, we create a new singleton segment. If the smallest existing segment is also a singleton, we merge the two segments and continue merging until the new segment is smaller than the smallest existing segment. This merging is controlled by the bits of the size field. If the lowest bit of size is zero, then we simply cons the new segment onto the segment list. If the lowest bit is one, then we merge the two segments and repeat. Of course, all this is done lazily.

```
fun add (x, (size, segs)) =
      let fun addSeg (seg, segs, size) =
                if size mod 2 = 0 then seg :: segs
                else addSeg (mrg (seg, hd segs), tl segs, size div 2)
      in (size+1, $addSeg ([x], force segs, size)) end
```

Finally, to sort a collection, we merge the segments from smallest to largest.

```
fun sort (size, segs) =
      let fun mrgAll (xs, [ ]) = xs
                | mrgAll (xs, seg :: segs) = mrgAll (mrg (xs, seg), segs)
      in mrgAll ([ ], force segs) end
```

Remark mrgAll can be viewed as computing

$$[\,] \bowtie s_1 \bowtie \cdots \bowtie s_m$$

where s_i is the ith segment and $\bowtie$ is left-associative, infix notation for mrg. This is a specific instance of a very common program schema, which can be written

$$c \oplus x_1 \oplus \cdots \oplus x_m$$

for any c and left-associative $\oplus$. Other instances of this schema include summing a list of integers ($c = 0$ and $\oplus = +$) or finding the maximum of a list of natural numbers ($c = 0$ and $\oplus =$ max). One of the greatest strengths of functional languages is the ability to define schemas like this as *higher-order functions* (i.e., functions that take functions as arguments or return functions as results). For example, the above schema might be written

```
fun foldl (f, c, [ ]) = c
  | foldl (f, c, x :: xs) = foldl (f, f (c, x), xs)
```

Then sort could be written

```
fun sort (size, segs) = foldl (mrg, [ ], force segs)
```

◇

The complete code for this implementation of sortable collections appears in Figure 6.5.

We show that add takes $O(\log n)$ amortized time and sort takes $O(n)$ amortized time using the physicist's method. We begin by defining the potential function Ψ, which is completely determined by the size of the collection:

$$\Psi(n) = 2n - 2\sum_{i=0}^{\infty} b_i(n \bmod 2^i + 1)$$

where b_i is the ith bit of n. Note that $\Psi(n)$ is bounded above by $2n$ and that $\Psi(n) = 0$ exactly when $n = 2^k - 1$ for some k.

Remark This potential function can be a little intimidating. It arises from considering each segment to have a potential proportional to its own size minus the sizes of all the smaller segments. The intuition is that the potential of a segment starts out big and gets smaller as more elements are added to the collection, reaching zero at the point just before the segment in question is merged with another segment. However, note that you do not need to understand the origins of a potential function to be able to calculate with it. ◇

We first calculate the complete cost of add. Its unshared cost is one and its

```
functor BottomUpMergeSort (Element : ORDERED) : SORTABLE =
struct
  structure Elem = Element

  type Sortable = int × Elem.T list list susp

  fun mrg ([ ], ys) = ys
    | mrg (xs, [ ]) = xs
    | mrg (xs as x :: xs', ys as y :: ys') =
        if Elem.leq (x, y) then x :: mrg (xs', ys) else y :: mrg (xs, ys')

  val empty = (0, $[ ])
  fun add (x, (size, segs)) =
        let fun addSeg (seg, segs, size) =
                  if size mod 2 = 0 then seg :: segs
                  else addSeg (mrg (seg, hd segs), tl segs, size div 2)
        in (size+1, $addSeg ([x], force segs, size)) end
  fun sort (size, segs) =
        let fun mrgAll (xs, [ ]) = xs
                | mrgAll (xs, seg :: segs) = mrgAll (mrg (xs, seg), segs)
        in mrgAll ([ ], force segs) end
end
```

Figure 6.5. Sortable collections based on bottom-up mergesort.

shared cost is the cost of performing the merges in addSeg. Suppose that the lowest k bits of n are one (i.e., $b_i = 1$ for $i < k$ and $b_k = 0$). Then addSeg performs k merges. The first combines two lists of size 1, the second combines two lists of size 2, and so on. Since merging two lists of size m takes $2m$ steps, addSeg takes

$$(1+1)+(2+2)+\cdots+(2^{k-1}+2^{k-1}) = 2(\sum_{i=0}^{k-1} 2^i) = 2(2^k-1)$$

steps. The complete cost of add is therefore $2(2^k-1)+1 = 2^{k+1}-1$.

Next, we calculate the change in potential. Let $n' = n+1$ and let b'_i be the ith bit of n'. Then,

$$\begin{aligned}
&\Psi(n') - \Psi(n) \\
&= 2n' - 2\textstyle\sum_{i=0}^{\infty} b'_i(n' \bmod 2^i + 1) - (2n - 2\sum_{i=0}^{\infty} b_i(n \bmod 2^i + 1)) \\
&= 2 + 2\textstyle\sum_{i=0}^{\infty}(b_i(n \bmod 2^i + 1) - b'_i(n' \bmod 2^i + 1)) \\
&= 2 + 2\textstyle\sum_{i=0}^{\infty} \delta(i)
\end{aligned}$$

where $\delta(i) = b_i(n \bmod 2^i + 1) - b'_i(n' \bmod 2^i + 1)$. We consider three cases: $i < k$, $i = k$, and $i > k$.

- ($i < k$): Since $b_i = 1$ and $b'_i = 0, \delta(k) = n \bmod 2^i + 1$. But $n \bmod 2^i = 2^i - 1$ so $\delta(k) = 2^i$.
- ($i = k$): Since $b_k = 0$ and $b'_k = 1$, $\delta(k) = -(n' \bmod 2^k + 1)$. But $n' \bmod 2^k = 0$ so $\delta(k) = -1 = -b'_k$.
- ($i > k$): Since $b'_i = b_i$, $\delta(k) = b'_i(n \bmod 2^i - n' \bmod 2^i)$. But $n' \bmod 2^i = (n+1) \bmod 2^i = n \bmod 2^i + 1$ so $\delta(i) = b'_i(-1) = -b'_i$.

Therefore,

$$\begin{aligned} \Psi(n') - \Psi(n) &= 2 + 2\textstyle\sum_{i=0}^{\infty} \delta(i) \\ &= 2 + 2\textstyle\sum_{i=0}^{k-1} 2^i + 2\textstyle\sum_{i=k}^{\infty}(-b'_i) \\ &= 2 + 2(2^k - 1) - 2\textstyle\sum_{i=k}^{\infty} b'_i \\ &= 2^{k+1} - 2B' \end{aligned}$$

where B' is the number of one bits in n'. Then the amortized cost of add is

$$(2^{k+1} - 1) - (2^{k+1} - 2B') = 2B' - 1$$

Since B' is $O(\log n)$, so is the amortized cost of add.

Finally, we calculate the amortized cost of sort. The first action of sort is to force the suspended list of segments. Since the potential is not necessarily zero, this adds $\Psi(n)$ to the amortized cost of the operation. sort next merges the segments from smallest to largest. The worst case is when $n = 2^k - 1$, so that there is one segment of each size from 1 to 2^{k-1}. Merging these segments takes

$$\begin{aligned} &(1+2) + (1+2+4) + (1+2+4+8) + \cdots + (1+2+\cdots+2^{k-1}) \\ &= \sum_{i=1}^{k-1}\sum_{j=0}^{i} 2^j = \sum_{i=1}^{k-1}(2^{i+1} - 1) = (2^{k+1} - 4) - (k-1) = 2n - k - 1 \end{aligned}$$

steps altogether. The amortized cost of sort is therefore $O(n) + \Psi(n) = O(n)$.

Exercise 6.7 Change the representation from a suspended list of lists to a list of streams.

(a) Prove the bounds on add and sort using the banker's method.

(b) Write a function to extract the k smallest elements from a sortable collection. Prove that your function runs in no more than $O(k \log n)$ amortized time.

6.5 Lazy Pairing Heaps

Finally, we adapt the pairing heaps of Section 5.5 to cope with persistence. Unfortunately, analyzing the resulting data structure appears to be just as hard as analyzing the original. However, we conjecture that the new implementation is asymptotically as efficient in a persistent setting as the original implementation of pairing heaps is in an ephemeral setting.

Recall that, in the previous implementation of pairing heaps, the children of a node were represented as a Heap list. Deleting the minimum element threw away the root and then merged the children in pairs using the function

```
fun mergePairs [ ] = E
  | mergePairs [h] = h
  | mergePairs (h₁ :: h₂ :: hs) = merge (merge (h₁, h₂), mergePairs hs)
```

If we were to delete the minimum element of the same heap twice, mergePairs would be called twice, duplicating work and destroying any hope of amortized efficiency. To cope with persistence, we must prevent this duplicated work. We once again turn to lazy evaluation. Instead of a Heap list, we represent the children of a node as a Heap susp. The value of this suspension is equal to $mergePairs *cs*. Since mergePairs operates on pairs of children, we extend the suspension with two children at once. Therefore, we include an extra Heap field in each node to hold any partnerless children. If there are no partnerless children (i.e., if the number of children is even), then this extra field is empty. Since this field is in use only when the number of children is odd, we call it the *odd field*. The new datatype is thus

```
datatype Heap = E | T of Elem.T × Heap × Heap susp
```

The insert and findMin operations are almost unchanged.

```
fun insert (x, a) = merge (T (x, E, $E), a)
fun findMin (T (x, a, m)) = x
```

Previously, the merge operation was simple and the deleteMin operation was complex. Now, the situation is reversed—all the complexity of mergePairs has been shifted to merge, which sets up the appropriate suspensions. deleteMin simply forces the heap suspension and merges it with the odd field.

```
fun deleteMin (T (x, a, $b)) = merge (a, b)
```

We define merge in two steps. The first step checks for empty arguments and otherwise compares the two arguments to see which has the smaller root.

```
fun merge (a, E) = a
  | merge (E, b) = b
  | merge (a as T (x, _, _), b as T (y, _, _)) =
      if Elem.leq (x, y) then link (a, b) else link (b, a)
```

```
functor LazyPairingHeap (Element : ORDERED) : HEAP =
struct
  structure Elem = Element

  datatype Heap = E | T of Elem.T × Heap × Heap susp

  val empty = E
  fun isEmpty E = true | isEmpty _ = false

  fun merge (a, E) = a
    | merge (E, b) = b
    | merge (a as T (x, _, _), b as T (y, _, _)) =
        if Elem.leq (x, y) then link (a, b) else link (b, a)
  and link (T (x, E, m), a) = T (x, a, m)
    | link (T (x, b, m), a) = T (x, E, $merge (merge (a, b), force m))

  fun insert (x, a) = merge (T (x, E, $E), a)

  fun findMin E = raise EMPTY
    | findMin (T (x, a, m)) = x
  fun deleteMin E = raise EMPTY
    | deleteMin (T (x, a, $b)) = merge (a, b)
end
```

Figure 6.6. Persistent pairing heaps using lazy evaluation.

The second step, embodied in the link helper function, adds a new child to a node. If the odd field is empty, then this child is placed in the odd field.

```
fun link (T (x, E, m), a) = T (x, a, m)
```

Otherwise, the new child is paired with the child in the odd field, and both are added to the suspension. In other words, we extend the suspension m = \$mergePairs *cs* to \$mergePairs (*a* :: *b* :: *cs*). Observe that

```
$mergePairs (a :: b :: cs)
    ≡ $merge (merge (a, b), mergePairs cs)
    ≡ $merge (merge (a, b), force ($mergePairs cs))
    ≡ $merge (merge (a, b), force m)
```

so the second clause of link may be written

```
fun link (T (x, b, m), a) = T (x, E, $merge (merge (a, b), force m))
```

The complete code for this implementation appears in Figure 6.6.

Hint to Practitioners: Although it now deals gracefully with persistence, this implementation of pairing heaps is relatively slow in practice because of overheads associated with lazy evaluation. It shines, however, under heavily persistent usage, where we reap maximum benefit from memoization. It is also competitive in lazy languages, where all data structures pay the overheads of lazy evaluation regardless of whether they actually gain any benefit.

6.6 Chapter Notes

Debits Some analyses using the traditional banker's method, such as Tarjan's analysis of path compression [Tar83], include both credits and debits. Whenever an operation needs more credits than are currently available, it creates a credit–debit pair and immediately spends the credit. The debit remains as an obligation that must be fulfilled. Later, a surplus credit may be used to discharge the debit.† Any debits that remain at the end of the computation add to the total actual cost. Although there are some similarities between the two kinds of debits, there are also some clear differences. For instance, with the debits introduced in this chapter, any debits leftover at the end of the computation are silently discarded.

It is interesting that debits arise in Tarjan's analysis of path compression since path compression is essentially an application of memoization to the find function.

Amortization and Persistence Until this work, amortization and persistence were thought to be incompatible. Several researchers [DST94, Ram92] had noted that amortized data structures could not be made efficiently persistent using existing techniques for adding persistence to ephemeral data structures, such as [DSST89, Die89], for reasons similar to those cited in Section 5.6. Ironically, these techniques produce persistent data structures with amortized bounds, but the underlying data structure must be worst-case. (These techniques have other limitations as well. Most notably, they cannot be applied to data structures supporting functions that combine two or more versions. Examples of offending functions include list catenation and set union.)

The idea that lazy evaluation could reconcile amortization and persistence first appeared, in rudimentary form, in [Oka95c]. The theory and practice of this technique were further developed in [Oka95a, Oka96b].

† There is a clear analogy here to the spontaneous creation and mutual annihilation of particle–antiparticle pairs in physics. In fact, a better name for these debits might be "anticredits".

Amortization and Functional Data Structures In his thesis, Schoenmakers [Sch93] studies amortized data structures in a strict functional language, concentrating on formal derivations of amortized bounds using the traditional physicist's method. He avoids the problems of persistence by insisting that data structures only be used in a single-threaded fashion.

Queues and Binomial Heaps The queues in Section 6.3.2 and the lazy binomial heaps in Section 6.4.1 first appeared in [Oka96b]. The analysis of lazy binomial heaps can also be applied to King's implementation of binomial heaps [Kin94].

Time-Analysis of Lazy Programs Several researchers have developed theoretical frameworks for analyzing the time complexity of lazy programs [BH89, San90, San95, Wad88]. However, these frameworks are not yet mature enough to be useful in practice. One difficulty is that these frameworks are, in some ways, too general. In each of these systems, the cost of a program is calculated with respect to some context, which is a description of how the result of the program will be used. However, this approach is often inappropriate for a methodology of program development in which data structures are designed as abstract data types whose behavior, including time complexity, is specified in isolation. In contrast, our analyses prove results that are independent of context (i.e., that hold regardless of how the data structures are used).