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

7 - Eliminating Amortization pp. 83-98

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511530104.008

Cambridge University Press

7
Eliminating Amortization

Most of the time, we do not care whether a data structure has amortized bounds or worst-case bounds; our primary criteria for choosing one data structure over another are overall efficiency and simplicity of implementation (and perhaps availability of source code). However, in some application areas, it is important to bound the running times of individual operations, rather than sequences of operations. In these situations, a worst-case data structure will often be preferable to an amortized data structure, even if the amortized data structure is simpler and faster overall. Raman [Ram92] identifies several such application areas, including

- **Real-time systems:** In real-time systems, predictability is more important than raw speed [Sta88]. If an expensive operation causes the system to miss a hard deadline, it does not matter how many cheap operations finished well ahead of schedule.
- **Parallel systems:** If one processor in a synchronous system executes an expensive operation while the other processors execute cheap operations, then the other processors may sit idle until the slow processor finishes.
- **Interactive systems:** Interactive systems are similar to real-time systems — users often value consistency more than raw speed [But83]. For instance, users might prefer 100 1-second response times to 99 0.25-second response times and 1 25-second response time, even though the latter scenario is twice as fast.

Remark Raman also identified a fourth application area — persistent data structures. As discussed in the previous chapter, amortization was thought to be incompatible with persistence. But, of course, we now know this to be untrue. ◇

Does this mean that amortized data structures are of no interest to programmers in these areas? Not at all. Since amortized data structures are often simpler than worst-case data structures, it is sometimes easier to design an amortized data structure, and then convert it to a worst-case data structure, than to design a worst-case data structure from scratch.

In this chapter, we describe *scheduling* — a technique for converting lazy amortized data structures to worst-case data structures by systematically forcing lazy components in such a way that no suspension ever takes very long to execute. Scheduling extends every object with an extra component, called a *schedule*, that regulates the order in which the lazy components of that object are forced.

7.1 Scheduling

Amortized and worst-case data structures differ mainly in when the computations charged to a given operation occur. In a worst-case data structure, all computations charged to an operation occur during the operation. In an amortized data structure, some computations charged to an operation may actually occur during later operations. From this, we see that virtually all nominally worst-case data structures become amortized when implemented in an entirely lazy language because many computations are unnecessarily suspended. To describe true worst-case data structures, we therefore need a strict language. If we want to describe both amortized and worst-case data structures, we need a language that supports both lazy and strict evaluation. Given such a language, we can also consider an intriguing hybrid approach: worst-case data structures that use lazy evaluation internally. We obtain such data structures by beginning with lazy amortized data structures and modifying them in such a way that every operation runs in the allotted time.

In a lazy amortized data structure, any specific operation might take longer than the stated bounds. However, this only occurs when the operation forces a suspension that has been paid off, but that takes a long time to execute. To achieve worst-case bounds, we must guarantee that every suspension executes in no more than the allotted time.

Define the *intrinsic cost* of a suspension to be the amount of time it takes to force the suspension under the assumption that all other suspensions on which it depends have already been forced and memoized, and therefore each take only $O(1)$ time to execute. (This is similar to the definition of the unshared cost of an operation.) The first step in converting an amortized data structure to a worst-case data structure is to reduce the intrinsic cost of every suspension to less than the desired bounds. Usually, this involves rewriting expensive

monolithic functions to make them incremental, either by changing the underlying algorithms slightly or by switching from a representation that supports only monolithic functions, such as suspended lists, to one that supports incremental functions as well, such as streams.

Even if every suspension has a small intrinsic cost, some suspensions might still take longer than the allotted time to execute. This happens when one suspension depends on another suspension, which in turn depends on a third, and so on. If none of the suspensions have been executed previously, then forcing the first suspension results in a cascade of forces. For example, consider the following computation:

$$(\cdots((s_1 +\!\!+ s_2) +\!\!+ s_3) +\!\!+ \cdots) +\!\!+ s_k$$

Forcing the suspension returned by the outermost $+\!\!+$ triggers a chain reaction in which every $+\!\!+$ executes a single step. Even though the outermost suspension has an $O(1)$ intrinsic cost, the total time required to force this suspension is $O(k)$ (or even more if the first node of s_1 is expensive to force for some other reason).

Remark Have you ever stood dominoes in a row so that each one knocks over the next? Even though the intrinsic cost of knocking over each domino is $O(1)$, the actual cost of knocking over the first domino might be much, much greater. ◇

The second step in converting an amortized data structure to a worst-case data structure is to avoid cascading forces by arranging that, whenever we force a suspension, any other suspensions on which it depends have already been forced and memoized. Then, no suspension takes longer than its intrinsic cost to execute. We accomplish this by systematically *scheduling* the execution of each suspension so that each is ready by the time we need it. The trick is to regard paying off debt as a literal activity, and to force each suspension as it is paid for.

Remark In essence, scheduling is like knocking over a series of dominoes starting from the rear, so that, whenever one domino falls on another, the second domino has already been knocked over. Then the actual cost of knocking over each domino is small. ◇

We extend every object with an extra component, called the *schedule*, that, at least conceptually, contains a pointer to every unevaluated suspension in the object. Some of the suspensions in the schedule may have already been evaluated in a different logical future, but forcing these suspensions a second time does no harm since it can only make an algorithm run faster than expected,

not slower. Every operation, in addition to whatever other manipulations it performs on an object, forces the first few suspensions in the schedule. The exact number of suspensions forced is governed by the amortized analysis; typically, every suspension takes $O(1)$ time to execute, so we force a number of suspensions proportional to the amortized cost of the operation. Depending on the data structure, maintaining the schedule can be non-trivial. For this technique to apply, adding a new suspension to the schedule, or retrieving the next suspension to be forced, cannot require more time than the desired worst-case bounds.

7.2 Real-Time Queues

As an example of this technique, we convert the amortized banker's queues of Section 6.3.2 to worst-case queues. Queues such as these that support all operations in $O(1)$ worst-case time are called *real-time queues* [HM81].

In the original data structure, queues are rotated using ++ and reverse. Since reverse is monolithic, our first task is finding a way to perform rotations incrementally. This can be done by executing one step of the reverse for every step of the ++. We define a function rotate such that

$$\text{rotate } (xs, ys, a) \equiv xs \mathbin{+\!\!+} \text{reverse } ys \mathbin{+\!\!+} a$$

Then

$$\text{rotate } (f, r, \$\text{NIL}) \equiv f \mathbin{+\!\!+} \text{reverse } r$$

The extra argument, a, is called an *accumulating parameter* and is used to accumulate the partial results of reversing ys. It is initially empty.

Rotations occur when $|r| = |f| + 1$, so initially $|ys| = |xs| + 1$. This relationship is preserved throughout the rotation, so when xs is empty, ys contains a single element. The base case is therefore

$$\begin{aligned}&\text{rotate } (\$\text{NIL}, \$\text{CONS } (y, \$\text{NIL}), a)\\ &\quad \equiv (\$\text{NIL}) \mathbin{+\!\!+} \text{reverse } (\$\text{CONS } (y, \$\text{NIL})) \mathbin{+\!\!+} a\\ &\quad \equiv \$\text{CONS } (y, a)\end{aligned}$$

In the recursive case,

$$\begin{aligned}&\text{rotate } (\$\text{CONS } (x, xs), \$\text{CONS } (y, ys), a)\\ &\quad \equiv (\$\text{CONS } (x, xs)) \mathbin{+\!\!+} \text{reverse } (\$\text{CONS } (y, ys)) \mathbin{+\!\!+} a\\ &\quad \equiv \$\text{CONS } (x, xs \mathbin{+\!\!+} \text{reverse } (\$\text{CONS } (y, ys)) \mathbin{+\!\!+} a)\\ &\quad \equiv \$\text{CONS } (x, xs \mathbin{+\!\!+} \text{reverse } ys \mathbin{+\!\!+} \$\text{CONS } (y, a))\\ &\quad \equiv \$\text{CONS } (x, \text{rotate } (xs, ys, \$\text{CONS } (y, a)))\end{aligned}$$

Putting these cases together, we get

```
fun rotate ($NIL, $CONS (y, _), a) = $CONS (y, a)
  | rotate ($CONS (x, xs), $CONS (y, ys), a) =
      $CONS (x, rotate (xs, ys, $CONS (y, a)))
```

Note that the intrinsic cost of every suspension created by rotate is $O(1)$.

Exercise 7.1 Show that replacing f ++ reverse r with rotate (f, r, $NIL) in the banker's queues of Section 6.3.2 reduces the worst-case running times of snoc, head, and tail from $O(n)$ to $O(\log n)$. (Hint: Prove that the longest chain of dependencies between suspensions is $O(\log n)$.) If it makes your analysis simpler, you may delay the pattern matching in the rotate function by writing **fun lazy** instead of **fun** . ◇

Next, we add a schedule to the datatype. The original datatype was

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

We extend this type with a new field s of type α Stream that represents a schedule for forcing the nodes of f. We can think of s in two ways, either as a suffix of f or as a pointer to the first unevaluated suspension in f. To evaluate the next suspension in the schedule, we simply force s.

Besides adding s, we make two further changes to the datatype. First, to emphasize the fact that the nodes of r need not be scheduled, we change r from a stream to a list. This involves minor changes to rotate. Second, we eliminate the length fields. As we will see shortly, we no longer need the length fields to determine when r becomes longer than f — instead, we can obtain this information from the schedule. The new datatype is thus

```
type α Queue = α Stream × α list × α Stream
```

Remark The savings in space from using three-tuples instead of four-tuples can make this change in representation worthwhile even if we don't care about worst-case bounds. ◇

With this representation, the major queue functions are simply

```
fun snoc ((f, r, s), x) = exec (f, x :: r, s)
fun head ($CONS (x, f), r, s) = x
fun tail ($CONS (x, f), r, s) = exec (f, r, s)
```

The helper function exec executes the next suspension in the schedule and maintains the invariant that $|s| = |f| - |r|$ (which incidentally guarantees that $|f| \geq |r|$ since $|s|$ cannot be negative). snoc increases $|r|$ by one and tail decreases $|f|$ by one, so when exec is called, $|s| = |f| - |r| + 1$. If s is non-empty, then we restore the invariant simply by taking the tail of s. If s is empty, then

```
structure RealTimeQueue : QUEUE =
struct
  type α Queue = α Stream × α list × α Stream

  val empty = ($NIL, [], $NIL)
  fun isEmpty ($NIL, _, _) = true
    | isEmpty _ = false

  fun rotate ($NIL, y :: _, a) = $CONS (y, a)
    | rotate ($CONS (x, xs), y :: ys, a) =
        $CONS (x, rotate (xs, ys, $CONS (y, a)))

  fun exec (f, r, $CONS (x, s)) = (f, r, s)
    | exec (f, r, $NIL) = let val f' = rotate (f, r, $NIL) in (f', [], f') end

  fun snoc ((f, r, s), x) = exec (f, x :: r, s)

  fun head ($NIL, r, s) = raise EMPTY
    | head ($CONS (x, f), r, s) = x
  fun tail ($NIL, r, s) = raise EMPTY
    | tail ($CONS (x, f), r, s) = exec (f, r, s)
end
```

Figure 7.1. Real-time queues based on scheduling.

r is one longer than f, so we rotate the queue. In either case, the very act of pattern matching against s to determine whether or not it is empty forces and memoizes the next suspension in the schedule.

```
fun exec (f, r, $CONS (x, s)) = (f, r, s)
  | exec (f, r, $NIL) = let val f' = rotate (f, r, $NIL) in (f', [], f') end
```

The complete code for this implementation appears in Figure 7.1.

By inspection, every queue operation does only $O(1)$ work outside of forcing suspensions, and no operation forces more than three suspensions. Hence, to show that all queue operations run in $O(1)$ worst-case time, we must prove that no suspension takes more than $O(1)$ time to execute.

Only three forms of suspensions are created by the various queue functions.

- $NIL is created by empty and exec (in the initial call to rotate). This suspension is trivial and therefore executes in $O(1)$ time regardless of whether it has been forced and memoized previously.
- $CONS (*y*, *a*) is created in both lines of rotate and is also trivial.
- $CONS (*x*, rotate (*xs*, *ys*, $CONS (*y*, *a*))) is created in the second line of rotate. This suspension allocates a CONS cell, builds a new suspension, and makes a recursive call to rotate, which pattern matches against the first node in *xs* and immediately creates another suspension. Of these

actions, only the force inherent in the pattern match has even the possibility of taking more than $O(1)$ time. But note that *xs* is a suffix of the front stream that existed just before the previous rotation. The treatment of the schedule *s* guarantees that *every* node in that stream was forced and memoized prior to the rotation, so forcing this node again takes only $O(1)$ time.

Since every suspension executes in $O(1)$ time, every queue operation runs in $O(1)$ worst-case time.

Hint to Practitioners: These queues are by far the simplest of the real-time implementations. They are also among the fastest known implementations—worst-case or amortized—for applications that use persistence heavily.

Exercise 7.2 Compute the size of a queue from the sizes of *s* and *r*. How much faster might such a function run than one that measures the sizes of *f* and *r*?

7.3 Binomial Heaps

We next return to the lazy binomial heaps from Section 6.4.1, and use scheduling to support insertions in $O(1)$ worst-case time. Recall that, in the earlier implementation, the representation of the heap was a Tree list susp, so insert was necessarily monolithic. Our first goal is to make insert incremental.

We begin by substituting streams for suspended lists in the type of heaps. The insert function calls the insTree helper function, which can now be written as follows:

```
fun lazy insTree (t, $NIL) = $CONS (t, $NIL)
       | insTree (t, ts as $CONS (t', ts')) =
           if rank t < rank t' then $CONS (t, ts)
           else insTree (link (t, t'), ts')
```

This function is still monolithic because it cannot return the first tree until it has performed all the links. To make this function incremental, we need a way for insTree to return a partial result after each iteration. We can achieve this by making the connection between binomial heaps and binary numbers more explicit. The trees in the heap correspond to the ones in the binary representation of the size of the heap. We extend this with an explicit representation of the zeros.

```
datatype Tree = NODE of Elem.T × Tree list
datatype Digit = ZERO | ONE of Tree
type Heap = Digit Stream
```

Note that we have eliminated the rank field in the NODE constructor because the rank of each tree is uniquely determined by its position: a tree in the ith digit has rank i, and the children of a rank r node have ranks $r-1, \ldots, 0$. In addition, we will insist that every non-empty digit stream end in a ONE.

Now insTree can be written

```
fun lazy insTree (t, $NIL) = $CONS (ONE t, $NIL)
       | insTree (t, $CONS (ZERO, ds)) = $CONS (ONE t, ds)
       | insTree (t, $CONS (ONE t', ds)) =
           $CONS (ZERO, insTree (link (t, t'), ds))
```

This function is properly incremental since each intermediate step returns a CONS cell containing a ZERO and a suspension for the rest of the computation. The final step always returns a ONE.

Next, we add a schedule to the datatype. The schedule is a list of jobs, where each job is a Digit Stream representing a call to insTree that has not yet been fully executed.

```
type Schedule = Digit Stream list
type Heap = Digit Stream × Schedule
```

To execute one step of the schedule, we force the head of the first job. If the result is a ONE, then this job is finished so we delete it from the schedule. If the result is a ZERO, then we put the rest of the job back in the schedule.

```
fun exec [] = []
  | exec (($CONS (ONE t, _)) :: sched) = sched
  | exec (($CONS (ZERO, job)) :: sched) = job :: sched
```

Finally, we update insert to maintain the schedule. Since the amortized cost of insert was two, we guess that executing two steps per insert will be enough to force every suspension by the time it is needed.

```
fun insert (x, (ds, sched)) =
      let val ds' = insTree (NODE (x, []), ds)
      in (ds', exec (exec (ds' :: sched))) end
```

To show that insert runs in $O(1)$ worst-case time, we need to show that exec runs in $O(1)$ worst-case time. In particular, we need to show that, whenever exec forces a suspension (by pattern matching against it), any other suspensions on which the first suspension depends have already been forced and memoized.

If we expand the **fun lazy** syntax in the definition of insTree and simplify slightly, we see that insTree produces a suspension equivalent to

```
$case ds of
    $NIL ⇒ CONS (ONE t, $NIL)
  | $CONS (ZERO, ds') ⇒ CONS (ONE t, ds')
  | $CONS (ONE t', ds') ⇒ CONS (ZERO, insTree (link (t, t'), ds'))
```

The suspension for each digit produced by insTree depends on the suspension for the previous digit at the same index. We prove that there is never more than one outstanding suspension per index of the digit stream and hence that no unevaluated suspension depends on another unevaluated suspension.

Define the *range* of a job in the schedule to be the collection of digits produced by the corresponding call to insTree. Each range comprises a possibly empty sequence of ZEROs followed by a ONE. We say that two ranges *overlap* if any of their digits have the same index within the stream of digits. Every unevaluated digit is in the range of some job in the schedule, so we need to prove that no two ranges overlap.

In fact, we prove a slightly stronger result. Define a *completed zero* to be a ZERO whose cell in the stream has already been evaluated and memoized.

Theorem 7.1 *Every valid heap contains at least two completed zeros prior to the first range in the schedule, and at least one completed zero between every two adjacent ranges in the schedule.*

Proof Let r_1 and r_2 be the first two ranges in the schedule. Let z_1 and z_2 be the two completed zeros before r_1, and let z_3 be the completed zero between r_1 and r_2. insert adds a new range r_0 to the front of the schedule and then immediately calls exec twice. Note that r_0 terminates in a ONE that replaces z_1. Let m be the number of ZEROs in r_0. There are three cases.

Case 1. $m = 0$. The only digit in r_0 is a ONE, so r_0 is eliminated by the first exec. The second exec forces the first digit of r_1. If this digit is ZERO, then it becomes the second completed zero (along with z_2) before the first range. If this digit is ONE, then r_1 is eliminated and r_2 becomes the new first range. The two completed zeros prior to r_2 are z_2 and z_3.

Case 2. $m = 1$. The two digits in r_0 are ZERO and ONE. These digits are immediately forced by the two execs, eliminating r_0. The leading ZERO replaces z_1 as one of the two completed zeros before r_1.

Case 3. $m \geq 2$. The first two digits of r_0 are both ZEROs. After the two calls to exec, these digits become the two completed zeros before the new first range (the rest of r_0). z_2 becomes the single completed zero between r_0 and r_1.

□

Exercise 7.3 Show that it does no harm to the running time of insert to remove the **lazy** annotation from the definition of insTree. ◇

Adapting the remaining functions to the new types is fairly straightforward. The complete implementation is shown in Figure 7.2. Four points about this code deserve further comment. First, rather than trying to do something clever with the schedule, merge and deleteMin evaluate every suspension in the system (using the function normalize) and set the schedule to []. Second, by Theorem 7.1, no heap contains more than $O(\log n)$ unevaluated suspensions, so forcing these suspensions during normalization or while searching for the minimum root does not affect the asymptotic running-times of merge, findMin, or deleteMin, each of which runs in $O(\log n)$ worst-case time. Third, the helper function removeMinTree sometimes produces digit streams with trailing ZEROs, but these streams are either discarded by findMin or merged with a list of ONEs by deleteMin. Finally, deleteMin must do a little more work than in previous implementations to convert a list of children into a valid heap. In addition to reversing the list, deleteMin must add a ONE to every tree and then convert the list to a stream. If *c* is the list of children, then this whole process can be written

```
listToStream (map ONE (rev c))
```

where

```
fun listToStream [] = $NIL
  | listToStream (x :: xs) = $CONS (x, listToStream xs)
fun map f [] = []
  | map f (x :: xs) = (f x) :: (map f xs)
```

map is the standard function for applying another function (in this case, the ONE constructor) to every element of a list.

Exercise 7.4 Write an efficient, specialized version of mrg, called mrgWithList, so that deleteMin can call

```
mrgWithList (rev c, ds')
```

instead of

```
mrg (listToStream (map ONE (rev c)), ds')
```

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

  datatype Tree = NODE of Elem.T × Tree list
  datatype Digit = ZERO | ONE of Tree
  type Schedule = Digit Stream list
  type Heap = Digit Stream × Schedule

  val empty = ($NIL, [])
  fun isEmpty ($NIL, _) = true | isEmpty _ = false

  fun link (t1 as NODE (x1, c1), t2 as NODE (x2, c2)) =
        if Elem.leq (x1,x2) then NODE (x1, t2 :: c1) else NODE (x2, t1 :: c2)
  fun insTree (t, $NIL) = $CONS (ONE t, $NIL)
    | insTree (t, $CONS (ZERO, ds)) = $CONS (ONE t, ds)
    | insTree (t, $CONS (ONE t', ds)) =
        $CONS (ZERO, insTree (link (t, t'), ds))
  fun mrg (ds1, $NIL) = ds1
    | mrg ($NIL, ds2) = ds2
    | mrg ($CONS (ZERO,ds1), $CONS (d,ds2)) = $CONS (d,mrg (ds1,ds2))
    | mrg ($CONS (d,ds1), $CONS (ZERO,ds2)) = $CONS (d,mrg (ds1,ds2))
    | mrg ($CONS (ONE t1, ds1), $CONS (ONE t2, ds2)) =
        $CONS (ZERO, insTree (link (t1, t2), mrg (ds1, ds2)))

  fun normalize (ds as $NIL) = ds
    | normalize (ds as $CONS (_, ds')) = (normalize ds'; ds)
  fun exec [] = []
    | exec (($CONS (ZERO, job)) :: sched) = job :: sched
    | exec (_ :: sched) = sched

  fun insert (x, (ds, sched)) =
        let val ds' = insTree (NODE (x, []), ds)
        in (ds', exec (exec (ds' :: sched))) end
  fun merge ((ds1, _), (ds2, _)) =
        let val ds = normalize (mrg (ds1, ds2)) in (ds, []) end

  fun removeMinTree ($NIL) = raise EMPTY
    | removeMinTree ($CONS (ONE t, $NIL)) = (t, $NIL)
    | removeMinTree ($CONS (ZERO, ds)) =
        let val (t',ds') = removeMinTree ds in (t',$CONS (ZERO,ds')) end
    | removeMinTree ($CONS (ONE (t as NODE (x, _)), ds)) =
        case removeMinTree ds of
          (t' as NODE (x', _), ds') ⇒
            if Elem.leq (x, x') then (t, $CONS (ZERO, ds))
            else (t', $CONS (ONE t, ds'))
  fun findMin (ds, _) =
        let val (NODE (x, _), _) = removeMinTree ds in x end
  fun deleteMin (ds, _) =
        let val (NODE (x, c), ds') = removeMinTree ds
            val ds'' = mrg (listToStream (map ONE (rev c)), ds')
        in (normalize ds'', []) end
end
```

Figure 7.2. Scheduled binomial heaps.

7.4 Bottom-Up Mergesort with Sharing

As a third example of scheduling, we modify the sortable collections from Section 6.4.3 to support add in $O(\log n)$ worst-case time and sort in $O(n)$ worst-case time.

The only use of lazy evaluation in the amortized implementation is the suspended call to addSeg in add. This suspension is monolithic, so the first task is to perform this computation incrementally. In fact, we need only make mrg incremental: since addSeg takes only $O(\log n)$ steps, we can afford to execute it strictly. We therefore represent segments as streams rather than lists, and eliminate the suspension on the collection of segments. The new type for the collection of segments is thus Elem.T Stream list rather than Elem.T list list susp.

Rewriting mrg, add, and sort to use this new type is straightforward, except that sort must convert the final sorted stream back to a list. This is accomplished by the streamToList conversion function.

```
fun streamToList ($NIL) = []
  | streamToList ($CONS (x, xs)) = x :: streamToList xs
```

The new version of mrg, shown in Figure 7.3, performs one step of the merge at a time, with an $O(1)$ intrinsic cost per step. Our second goal is to execute enough merge steps per add to guarantee that any sortable collection contains only $O(n)$ unevaluated suspensions. Then sort executes at most $O(n)$ unevaluated suspensions in addition to its own $O(n)$ work. Executing these unevaluated suspensions takes at most $O(n)$ time, so sort takes only $O(n)$ time altogether.

In the amortized analysis, the amortized cost of add was approximately $2B'$, where B' is the number of one bits in $n' = n+1$. This suggests that add should execute two suspensions per one bit, or equivalently, two suspensions per segment. We maintain a separate schedule for each segment. Each schedule is a list of streams, each of which represents a call to mrg that has not yet been fully evaluated. The complete type is therefore

```
type Schedule = Elem.T Stream list
type Sortable = int × (Elem.T Stream × Schedule) list
```

To execute one merge step from a schedule, we call the function exec1.

```
fun exec1 [] = []
  | exec1 (($NIL) :: sched) = exec1 sched
  | exec1 (($CONS (x, xs)) :: sched) = xs :: sched
```

In the second clause, we reach the end of one stream and execute the first step of the next stream. This cannot loop because only the first stream in a schedule can ever be empty. The function exec2 takes a segment and invokes exec1 twice on the schedule.

```
fun exec2 (xs, sched) = (xs, exec1 (exec1 sched))
```

Now, add calls exec2 on every segment, but it is also responsible for building the schedule for the new segment. If the lowest k bits of n are one, then adding a new element will trigger k merges, of the form

$$((s_0 \bowtie s_1) \bowtie s_2) \bowtie \cdots \bowtie s_k$$

where s_0 is the new singleton segment and $s_1 \ldots s_k$ are the first k segments of the existing collection. The partial results of this computation are $s'_1 \ldots s'_k$, where $s'_1 = s_0 \bowtie s_1$ and $s'_i = s'_{i-1} \bowtie s_i$. Since the suspensions in s'_i depend on the suspensions in s'_{i-1}, we must schedule the execution of s'_{i-1} before the execution of s'_i. The suspensions in s'_i also depend on the suspensions in s_i, but we guarantee that $s_1 \ldots s_k$ have been completely evaluated at the time of the call to add.

The final version of add, which creates the new schedule and executes two suspensions per segment, is

```
fun add (x, (size, segs)) =
    let fun addSeg (xs, segs, size, rsched) =
            if size mod 2 = 0 then (xs, rev rsched) :: segs
            else let val ((xs', []) :: segs') = segs
                     val xs'' = mrg (xs, xs')
                 in addSeg (xs'', segs', size div 2, xs'' :: rsched) end
        val segs' = addSeg ($CONS (x, $NIL), segs, size, [])
    in (size+1, map exec2 segs') end
```

The accumulating parameter *rsched* collects the newly merged streams in reverse order. Therefore, we reverse it back to the correct order on the last step. The pattern match in line 4 asserts that the old schedule for that segment is empty, i.e., that it has already been completely executed. We will see shortly why this true.

The complete code for this implementation is shown in Figure 7.3. add has an unshared cost of $O(\log n)$ and sort has an unshared cost of $O(n)$, so to prove the desired worst-case bounds, we must show that the $O(\log n)$ suspensions forced by add take $O(1)$ time each, and that the $O(n)$ unevaluated suspensions forced by sort take $O(n)$ time altogether.

Every merge step forced by add (through exec2 and exec1) depends on two other streams. If the current step is part of the stream s'_i, then it depends on the streams s'_{i-1} and s_i. The stream s'_{i-1} was scheduled before s'_i, so s'_{i-1} has been completely evaluated by the time we begin evaluating s'_i. Furthermore, s_i was completely evaluated before the add that created s'_i. Since the intrinsic cost of each merge step is $O(1)$, and the suspensions forced by each step have

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

  type Schedule = Elem.T Stream list
  type Sortable = int × (Elem.T Stream × Schedule) list

  fun lazy mrg ($NIL, ys) = ys
         | mrg (xs, $NIL) = xs
         | mrg (xs as $CONS (x, xs'), ys as $CONS (y, ys')) =
             if Elem.leq (x, y) then $CONS (x, mrg (xs', ys))
             else $CONS (y, mrg (xs, ys'))

  fun exec1 [] = []
    | exec1 (($NIL) :: sched) = exec1 sched
    | exec1 (($CONS (x, xs)) :: sched) = xs :: sched
  fun exec2 (xs, sched) = (xs, exec1 (exec1 sched))

  val empty = (0, [])
  fun add (x, (size, segs)) =
        let fun addSeg (xs, segs, size, rsched) =
                  if size mod 2 = 0 then (xs, rev rsched) :: segs
                  else let val ((xs', []) :: segs') = segs
                           val xs'' = mrg (xs, xs')
                       in addSeg (xs'', segs', size div 2, xs'' :: rsched)
            val segs' = addSeg ($CONS (x, $NIL), segs, size, [])
        in (size+1, map exec2 segs') end
  fun sort (size, segs) =
        let fun mrgAll (xs, []) = xs
              | mrgAll (xs, (xs', _) :: segs) = mrgAll (mrg (xs, xs'), segs)
        in streamToList (mrgAll ($NIL, segs)) end
end
```

Figure 7.3. Scheduled bottom-up mergesort.

already been forced and memoized, every merge step forced by add takes only $O(1)$ worst-case time.

The following lemma establishes both that any segment involved in a merge by addSeg has been completely evaluated and that the collection as a whole contains at most $O(n)$ unevaluated suspensions.

Lemma 7.2 *In any sortable collection of size n, the schedule for a segment of size $m = 2^k$ contains a total of at most $2m - 2(n \bmod m + 1)$ elements.*

Proof Consider a sortable collection of size n, where the lowest k bits of n are ones (i.e., n can be written $c2^{k+1} + (2^k - 1)$, for some integer c). Then add produces a new segment of size $m = 2^k$, whose schedule contains streams of sizes $2, 4, 8, \ldots, 2^k$. The total size of this schedule is $2^{k+1} - 2 = 2m - 2$. After

executing two steps, the size of the schedule is $2m - 4$. The size of the new collection is $n' = n + 1 = c2^{k+1} + 2^k$. Since $2m - 4 < 2m - 2(n' \bmod m + 1) = 2m - 2$, the lemma holds for this segment.

Every segment of size m' larger than m is unaffected by the add, except for the execution of two steps from the segment's schedule. The size of the new schedule is bounded by

$$2m' - 2(n \bmod m' + 1) - 2 = 2m' - 2(n' \bmod m' + 1),$$

so the lemma holds for these segments as well. □

Now, whenever the k lowest bits of n are ones (i.e., whenever the next add will merge the first k segments), we know by Lemma 7.2 that, for any segment of size $m = 2^i$, where $i < k$, the number of elements in that segment's schedule is at most

$$2m - 2(n \bmod m + 1) = 2m - 2((m - 1) + 1) = 0$$

In other words, that segment has been completely evaluated.

Finally, the combined schedules for all segments comprise at most

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

elements, where b_i is the ith bit of n. Note the similarity to the potential function from the physicist's analysis in Section 6.4.3. Since this total is bounded by $2n$, the collection as a whole contains only $O(n)$ unevaluated suspensions, and therefore sort runs in $O(n)$ worst-case time.

7.5 Chapter Notes

Eliminating Amortization Dietz and Raman [DR91, DR93, Ram92] have devised a framework for eliminating amortization based on *pebble games*, where the derived worst-case algorithms correspond to winning strategies in some game. Others have used ad hoc techniques similar to scheduling to eliminate amortization from specific data structures such as *implicit binomial queues* [CMP88] and *relaxed heaps* [DGST88]. The form of scheduling described here was first applied to queues in [Oka95c] and later generalized in [Oka96b].

Queues The queue implementation in Section 7.2 first appeared in [Oka95c]. Hood and Melville [HM81] presented the first purely functional implementation of real-time queues, based on a technique known as *global rebuild-*

ing [Ove83], which will be discussed further in the next chapter. Their implementation does not use lazy evaluation and is more complicated than ours.