Amortized Complexity Verified

Tobias Nipkow(✉)

Technische Universität München, Munich, Germany
nipkow@in.tum.de

Abstract. A framework for the analysis of the amortized complexity of (functional) data structures is formalized in Isabelle/HOL and applied to a number of standard examples and to three famous non-trivial ones: skew heaps, splay trees and splay heaps.

1 Introduction

Amortized complexity [3,14] of an algorithm averages the running times of a sequence of invocations of the algorithm. In this paper we formalize a simple framework for the analysis of the amortized complexity of functional programs and apply it to both the easy standard examples and the more challenging examples of skew heaps, splay trees and splay heaps. We have also analyzed pairing heaps [4] but cannot present them here for lack of space. All proofs are available online [9].

We are aiming for a particularly lightweight framework that supports proofs at a high level of abstraction. Therefore all algorithms are modeled as recursive functions in the logic. Because mathematical functions do not have a complexity they need to be accompanied by definitions of the intended running time that follow the recursive structure of the actual function definitions. Thus the user is free to choose the level of granularity of the complexity analysis. In our examples we simply count function calls.

Although one can view the artefacts that we analyze as functional programs, one can also view them as models or abstractions of imperative programs. For the amortized complexity we are only interested in the input-output behaviour and the running time complexity. As long as those are the same, it does not matter in what language the algorithm is implemented. In fact, the standard imperative implementations of all of our examples have the same complexity as the functional model. However, in a functional setting, amortized complexity reasoning may be invalid if the data structure under consideration is not used in a single-threaded manner [10].

1.1 Related Work

Hofmann and Jost [7] pioneered automatic type-based amortized analysis of heap usage of functional programs. This was later generalized in many directions, for example allowing multivariate polynomials [6]. Atkey [1] carries some of the

C. Urban and X. Zhang (Eds.): ITP 2015, LNCS 9236, pp. 310–324, 2015.
DOI: 10.1007/978-3-319-22102-1_21

ideas over to an imperative language with separation logic embedded in Coq. Charguéraud and Pottier [2] employ separation logic to verify the almost-linear amortized complexity of a Union-Find implementation in OCaml in Coq.

2 Lists and Trees

Lists are constructed from the empty list [] via the infix cons-operator "$\cdot$", $|xs|$ is the length of xs, tl takes the tail and rev reverses a list.

Binary trees are defined as the data type $'a\ tree$ with two constructors: the empty tree or leaf $\langle\rangle$ and the node $\langle l, a, r\rangle$ with subtrees $l, r :: 'a\ tree$ and contents $a :: 'a$. The size of a tree is the number of its nodes:

$$|\langle\rangle| = 0 \qquad |\langle l, \_, r\rangle| = |l| + |r| + 1$$

For convenience there is also the modified size function $|t|_1 = |t| + 1$.

3 Amortized Analysis Formalized

We formalize the following scenario. We are given a number of operations that may take parameters and that update the state (some data structure) as a side effect. Our model is purely functional: the state is replaced by a new state with each invocation of an operation, rather than mutating the state. This makes no difference because we only analyze time, not space.

Our model of amortized analysis is a theory that is parameterized as follows (a *locale* in Isabelle-speak):

$'s$ is the type of state.
$'o$ is the type of operations.
$init :: 's$ is the initial state.
$nxt :: 'o \Rightarrow 's \Rightarrow 's$ is the next state function.
$inv :: 's \Rightarrow bool$ is an invariant on the state. We assume that the invariant holds initially ($inv\ init$) and that it is preserved by all operations ($inv\ s \Longrightarrow inv\ (nxt\ f\ s)$).
$t :: 'o \Rightarrow 's \Rightarrow real$ is the timing function: $t\ f\ s$ represents the time it takes to execute operation f in state s, i.e. $nxt\ f\ s$.

The effect of each operation f is modeled as a function $nxt\ f$ from state to state. Since functions are extensional, the execution time is modeled explicitly by function t. Alternatively one can instrument nxt with timing information and have it return a pair of a new state and the time the operation took. We have separated the computation of the result and the timing information into two functions because that is what one would typically do in a first step anyway to simplify the proofs. In particular this means that t need not be a closed form expression for the actual complexity. In all of our examples the definition of t will follow the (usually recursive) structure of the definition of nxt precisely. One could go one step further and derive t from an intensional formulation of nxt

automatically, but that is orthogonal to our aim in this paper, namely *amortized* complexity.

For the analysis of amortized complexity we formalize the *potential method*. That is, our theory has another parameter:

$\Phi :: 's \Rightarrow real$ is the *potential* of a state. We assume the potential is initially 0 ($\Phi\ init = 0$) and never becomes negative ($inv\ s \Longrightarrow 0 \leq \Phi\ s$).

The potential of the state represents the savings that can pay for future restructurings of the data structure. Typically, the higher the potential, the more out of balance the data structure is. Note that the potential is just a means to an end, the analysis, but does not influence the actual operations.

Let us now analyze the complexity of a sequence of operations formalized as a function of type $nat \Rightarrow 'o$. The sequence is infinite for convenience but of course we only analyze the execution of finite prefixes. For this purpose we define *state f n*, the state after the execution of the first n elements of f:

$state :: (nat \Rightarrow 'o) \Rightarrow nat \Rightarrow 's$
$state\ f\ 0 = init$
$state\ f\ (Suc\ n) = nxt\ (f\ n)\ (state\ f\ n)$

Now we can define the amortized complexity of an operation as the actual time it takes plus the difference in potential:

$a :: (nat \Rightarrow 'o) \Rightarrow nat \Rightarrow real$
$a\ f\ i = t\ (f\ i)\ (state\ f\ i) + \Phi\ (state\ f\ (i + 1)) - \Phi\ (state\ f\ i)$

By telescoping (i.e. induction) we obtain

$(\sum_{i<n}\ t\ (f\ i)\ (state\ f\ i)) = (\sum_{i<n}\ a\ f\ i) + \Phi\ (state\ f\ 0) - \Phi\ (state\ f\ n)$

where $\sum_{i<n}\ F\ i$ is the sum of all $F\ i$ with $i < n$. Because of the assumptions on Φ this implies that on average the amortized complexity is an upper bound of the real complexity:

$(\sum_{i<n}\ t\ (f\ i)\ (state\ f\ i)) \leq (\sum_{i<n}\ a\ f\ i)$

To complete our formalization we add one more parameter:

$U :: 'o \Rightarrow 's \Rightarrow real$ is an explicit upper bound for the amortized complexity of each operation (in a certain state), i.e. we assume that $inv\ s \Longrightarrow t\ f\ s + \Phi\ (nxt\ f\ s) - \Phi\ s \leq U\ f\ s$.

Thus we obtain that U is indeed an upper bound of the real complexity:

$(\sum_{i<n}\ t\ (f\ i)\ (state\ f\ i)) \leq (\sum_{i<n}\ U\ (f\ i)\ (state\ f\ i))$

Instantiating this theory of amortized complexity means defining the parameters and proving the assumptions, in particular about U.

4 Easy Examples

Unless noted otherwise, the examples in this section come from a standard textbook [3].

4.1 Binary Counter

We start with the binary counter explained in the introduction. The state space $'s$ is just a list of booleans, starting with the least significant bit. There is just one parameterless operation "increment". Thus we can model type $'o$ with the unit type. The increment operation is defined recursively:

$$\begin{array}{l} incr\ [] = [True] \\ incr\ (False \cdot bs) = True \cdot bs \\ incr\ (True \cdot bs) = False \cdot incr\ bs \end{array}$$

In complete analogy the running time function for *incr* is defined:

$$\begin{array}{l} t_{incr}\ [] = 1 \\ t_{incr}\ (False \cdot bs) = 1 \\ t_{incr}\ (True \cdot bs) = t_{incr}\ bs + 1 \end{array}$$

Now we can instantiate the parameters of the amortized analysis theory:

$$\begin{array}{lll} init = [] & nxt\ () = incr & t\ () = t_{incr} \\ inv\ s = True & \Phi\ s = |filter\ id\ s| & U\ ()\ s = 2 \end{array}$$

The key idea of the analysis is to define the potential of s as $|filter\ id\ s|$, the number of *True* bits in s. This makes sense because the higher the potential, the longer an increment may take (roughly speaking). Now it is easy to show that 2 is an upper bound for the amortized complexity: the requirement on U follows immediately from this lemma (which is proved by induction):

$$t_{incr}\ bs + \Phi_{incr}\ (incr\ bs) - \Phi_{incr}\ bs = 2$$

4.2 Stack with Multipop

The operations are

$$\textbf{datatype}\ 'a\ op_{stk} = Push\ 'a \mid Pop\ nat$$

where $Pop\ n$ pops n elements off the stack:

$$\begin{array}{l} nxt_{stk}\ (Push\ x)\ xs = x \cdot xs \\ nxt_{stk}\ (Pop\ n)\ xs = drop\ n\ xs \end{array}$$

In complete analogy the running time function is defined:

$$\begin{array}{l} t_{stk}\ (Push\ x)\ xs = 1 \\ t_{stk}\ (Pop\ n)\ xs = min\ n\ |xs| \end{array}$$

Now we can instantiate the parameters of the amortized analysis theory:

$$\begin{array}{llll} init = [] & nxt = nxt_{stk} & t = t_{stk} & inv\ s = True \\ \Phi = length & U\ f\ s = (\mathsf{case}\ f\ \mathsf{of}\ Push\ x \Rightarrow 2 \mid Pop\ n \Rightarrow 0) & & \end{array}$$

The necessary proofs are all automatic.

4.3 Dynamic Tables

Dynamic tables are tables where elements are added and deleted and the table grows and shrinks accordingly. We ignore the actual elements because they are irrelevant for the complexity analysis. Therefore the operations

datatype $op_{tb} = Ins \mid Del$

do not have arguments. Similarly the state is merely a pair of natural numbers $(n,\ l)$ that abstracts a table of size l with n elements. This is how the operations behave:

$nxt_{tb}\ Ins\ (n,\ l) = (n + 1,\ \mathit{if}\ n < l\ \mathit{then}\ l\ \mathit{else}\ \mathit{if}\ l = 0\ \mathit{then}\ 1\ \mathit{else}\ 2 * l)$
$nxt_{tb}\ Del\ (n,\ l) =$
$(n - 1,\ \mathit{if}\ n = 1\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ 4 * (n - 1) < l\ \mathit{then}\ l\ \mathit{div}\ 2\ \mathit{else}\ l)$

If the table overflows upon insertion, its size is doubled. If a table is less than one quarter full after deletion, its size is halved. The transition from and to the empty table is treated specially.

This is the corresponding running time function:

$t_{tb}\ Ins\ (n,\ l) = (\mathit{if}\ n < l\ \mathit{then}\ 1\ \mathit{else}\ n + 1)$
$t_{tb}\ Del\ (n,\ l) = (\mathit{if}\ n = 1\ \mathit{then}\ 1\ \mathit{else}\ \mathit{if}\ 4 * (n - 1) < l\ \mathit{then}\ n\ \mathit{else}\ 1)$

The running time for the cases where the table expands or shrinks is determined by the number of elements that need to be copied.

Now we can instantiate the parameters of the amortized analysis theory. We start with the system itself:

$init = (0,\ 0) \qquad nxt = nxt_{tb} \qquad t = t_{tb}$
$inv\ (n,l) = (\mathit{if}\ l = 0\ \mathit{then}\ n = 0\ \mathit{else}\ n \le l \wedge l \le 4 * n)$

This is the first time we have a non-trivial invariant. The potential is also more complicated than before:

$\Phi\ (n,l) = (\mathit{if}\ 2 * n < l\ \mathit{then}\ l\ /\ 2 - n\ \mathit{else}\ 2 * n - l)$

Now it is automatic to show the following amortized complexity:

$U\ f\ s = (\mathit{case}\ f\ \mathit{of}\ Ins \Rightarrow 3 \mid Del \Rightarrow 2)$

4.4 Queues

Queues have one operation for enqueueing a new item and one for dequeueing the oldest item:

datatype $'a\ op_q = Enq\ 'a \mid Deq$

We ignore accessing the oldest item because it is a constant time operation in our implementation.

The simplest possible implementation of functional queues (e.g. [10]) consists of two lists (stacks) $(xs,\ ys)$:

$$nxt_q\ (Enq\ x)\ (xs,\ ys) = (x \cdot xs,\ ys)$$
$$nxt_q\ Deq\ (xs,\ ys) = (\mathit{if}\ ys = []\ \mathit{then}\ ([],\ tl\ (rev\ xs))\ \mathit{else}\ (xs,\ tl\ ys))$$

$$t_q\ (Enq\ x)\ (xs,\ ys) = 1$$
$$t_q\ Deq\ (xs,\ ys) = (\mathit{if}\ ys = []\ \mathit{then}\ |xs|\ \mathit{else}\ 0)$$

Note that the running time function counts only allocations of list cells and that it assumes *rev* is linear. Now we can instantiate the parameters of the amortized analysis theory to show that the average complexity of both *Enq* and *Deq* is 2. The necessary proofs are all automatic.

$$init = ([],\ [])\qquad nxt = nxt_q\qquad t = t_q\qquad inv\ s = True$$
$$\Phi\ (xs,ys) = |xs|\qquad U\ f\ s = (\mathit{case}\ f\ \mathit{of}\ Enq\ x \Rightarrow 2 \mid Deq \Rightarrow 0)$$

In the same manner I have also verified [9] a modified implementation where reversal happens already when $|xs| = |ys| + 1$; this improves the worst-case behaviour but (using $\Phi(xs,ys) = 2 * |xs|$) the amortized complexity of *Enq* increases to 3.

5 Skew Heaps

This section analyzes a beautifully simple data structure for priority queues: skew heaps [13]. Heaps are trees where the least element is at the root. We assume that the elements are linearly ordered. The central operation on skew heaps is *meld*, that merges two skew heaps and swaps children along the merge path:

```
meld h1 h2 =
(case h1 of ⟨⟩ ⇒ h2
 | ⟨l1, a1, r1⟩ ⇒
    case h2 of ⟨⟩ ⇒ h1
    | ⟨l2, a2, r2⟩ ⇒
       if a1 ≤ a2 then ⟨meld h2 r1, a1, l1⟩ else ⟨meld h1 r2, a2, l2⟩)
```

We consider the two operations of inserting an element and removing the minimal element:

$$\textbf{datatype}\ 'a\ op_{pq} = Insert\ 'a \mid Delmin$$

They are implemented via *meld* as follows:

$$nxt_{pq}\ (Insert\ a)\ h = meld\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle\ h$$
$$nxt_{pq}\ Delmin\ h = del\_min\ h$$

$$del\_min\ \langle\rangle = \langle\rangle$$
$$del\_min\ \langle l,\ m,\ r\rangle = meld\ l\ r$$

For the functional correctness proofs see [9].

The analysis by Sleator and Tarjan is not ideal as a starting point for a formalization. Luckily there is a nice, precise functional account by Kaldewaij and Schoenmakers [8] that we will follow (although their *meld* differs slightly from ours). Their cost measure counts the number of calls of *meld*, *Insert* and *Delmin*:

$$t_{meld}\ \langle\rangle\ h = 1$$
$$t_{meld}\ h\ \langle\rangle = 1$$
$$t_{meld}\ \langle l_1,\ a_1,\ r_1\rangle\ \langle l_2,\ a_2,\ r_2\rangle =$$
$$(\mathit{if}\ a_1 \le a_2\ \mathit{then}\ t_{meld}\ \langle l_2,\ a_2,\ r_2\rangle\ r_1\ \mathit{else}\ t_{meld}\ \langle l_1,\ a_1,\ r_1\rangle\ r_2) + 1$$

$$t_{pq}\ (\mathit{Insert}\ a)\ h = t_{meld}\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle\ h + 1$$
$$t_{pq}\ \mathit{Delmin}\ h = (\mathit{case}\ h\ \mathit{of}\ \langle\rangle \Rightarrow 1 \mid \langle t_1,\ a,\ t_2\rangle \Rightarrow t_{meld}\ t_1\ t_2 + 1)$$

Kaldewaij and Schoenmakers prove a tighter upper bound than Sleator and Tarjan, replacing the factor of 3 by 1.44. We are satisfied with verifying the bound by Sleator and Tarjan and work with the following simple potential function which is an instance of the one by Kaldewaij and Schoenmakers: it counts the number of "right heavy" nodes.

$$\Phi\ \langle\rangle = 0$$
$$\Phi\ \langle l,\ \_,\ r\rangle = \Phi\ l + \Phi\ r + (\mathit{if}\ |l| < |r|\ \mathit{then}\ 1\ \mathit{else}\ 0)$$

To prove the amortized complexity of *meld* we need some further notions that capture the ideas of Sleator and Tarjan in a concise manner:

$$\mathit{rheavy}\ \langle l,\ \_,\ r\rangle = (|l| < |r|)$$

$$\mathit{lpath}\ \langle\rangle = []$$
$$\mathit{lpath}\ \langle l,\ a,\ r\rangle = \langle l,\ a,\ r\rangle \cdot \mathit{lpath}\ l$$

$$\mathit{rpath}\ \langle\rangle = []$$
$$\mathit{rpath}\ \langle l,\ a,\ r\rangle = \langle l,\ a,\ r\rangle \cdot \mathit{rpath}\ r$$

$$\Gamma\ h = |\mathit{filter}\ \mathit{rheavy}\ (\mathit{lpath}\ h)|$$
$$\Delta\ h = |\mathit{filter}\ (\lambda p.\ \neg\ \mathit{rheavy}\ p)\ (\mathit{rpath}\ h)|$$

Two easy inductive properties:

$$\Gamma\ h \le \log_2\ |h|_1 \quad (1) \qquad \Delta\ h \le \log_2\ |h|_1 \quad (2)$$

Now the desired logarithmic amortized complexity of *meld* follows:

$$\begin{aligned}
&t_{meld}\ t_1\ t_2 + \Phi\ (\mathit{meld}\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \\
&\le \Gamma\ (\mathit{meld}\ t_1\ t_2) + \Delta\ t_1 + \Delta\ t_2 + 1 \quad \text{by induction on } \mathit{meld} \\
&\le \log_2\ |\mathit{meld}\ t_1\ t_2|_1 + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \qquad \text{by (1), (2)} \\
&= \log_2\ (|t_1|_1 + |t_2|_1 - 1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
&\quad \text{because } |\mathit{meld}\ t_1\ t_2| = |t_1| + |t_2| \\
&\le \log_2\ (|t_1|_1 + |t_2|_1) + \log_2\ |t_1|_1 + \log_2\ |t_2|_1 + 1 \\
&\le \log_2\ (|t_1|_1 + |t_2|_1) + 2 * \log_2\ (|t_1|_1 + |t_2|_1) + 1 \\
&\quad \text{because } \log_2\ x + \log_2\ y \le 2 * \log_2\ (x + y) \text{ if } x, y \ge 1 \\
&= 3 * \log_2\ (|t_1|_1 + |t_2|_1) + 1
\end{aligned}$$

Now it is easy to verify the following amortized complexity for *Insert* and *Delmin* by instantiating our standard theory with

$$U\ (\mathit{Insert}\ \_)\ h = 3 * \log_2\ (|h|_1 + 2) + 2$$
$$U\ \mathit{Delmin} = 3 * \log_2\ (|h|_1 + 2) + 4$$

Note that Isabelle supports implicit coercions, in particular from *nat* to *real*, that are inserted automatically [15].

6 Splay Trees

A splay tree [12] is a subtle self-adjusting binary search tree. It achieves its amortized logarithmic complexity by local rotations of subtrees along the access path. Its central operation is *splay* of type $'a \Rightarrow 'a\ tree \Rightarrow 'a\ tree$ that rotates the given element (of a linearly ordered type $'a$) to the root of the tree. Most presentations of *splay* confine themselves to this case where the given element is in the tree. If the given element is not in the tree, the last element found before a $\langle\rangle$ was met is rotated to the root. The complete definition is shown in Fig. 1.

Given splaying, searching for an element in the tree is trivial: you splay with the given element and check if it ends up at the root. For insertion and deletion, algorithm texts typically show pictures only. In contrast, we show the code only, in Figs. 2–3. To insert a, you splay with a to see if it is already there, and if it is not, you insert it at the top (which is the right place due to the previous splay action).

$$
\begin{array}{l}
splay\ a\ \langle\rangle = \langle\rangle \\
splay\ a\ \langle cl,\ c,\ cr\rangle = \\
(\mathsf{if}\ a = c\ \mathsf{then}\ \langle cl,\ c,\ cr\rangle \\
\ \mathsf{else\ if}\ a < c \\
\quad \mathsf{then\ case}\ cl\ \mathsf{of}\ \langle\rangle \Rightarrow \langle cl,\ c,\ cr\rangle \\
\qquad \mid \langle bl,\ b,\ br\rangle \Rightarrow \\
\qquad\quad \mathsf{if}\ a = b\ \mathsf{then}\ \langle bl,\ a,\ \langle br,\ c,\ cr\rangle\rangle \\
\qquad\quad \mathsf{else\ if}\ a < b \\
\qquad\qquad \mathsf{then\ if}\ bl = \langle\rangle\ \mathsf{then}\ \langle bl,\ b,\ \langle br,\ c,\ cr\rangle\rangle \\
\qquad\qquad\quad \mathsf{else\ case}\ splay\ a\ bl\ \mathsf{of} \\
\qquad\qquad\qquad \langle al,\ a',\ ar\rangle \Rightarrow \langle al,\ a',\ \langle ar,\ b,\ \langle br,\ c,\ cr\rangle\rangle\rangle \\
\qquad\qquad \mathsf{else\ if}\ br = \langle\rangle\ \mathsf{then}\ \langle bl,\ b,\ \langle br,\ c,\ cr\rangle\rangle \\
\qquad\qquad\quad \mathsf{else\ case}\ splay\ a\ br\ \mathsf{of} \\
\qquad\qquad\qquad \langle al,\ a',\ ar\rangle \Rightarrow \langle\langle bl,\ b,\ al\rangle,\ a',\ \langle ar,\ c,\ cr\rangle\rangle \\
\quad \mathsf{else\ case}\ cr\ \mathsf{of}\ \langle\rangle \Rightarrow \langle cl,\ c,\ cr\rangle \\
\qquad \mid \langle bl,\ b,\ br\rangle \Rightarrow \\
\qquad\quad \mathsf{if}\ a = b\ \mathsf{then}\ \langle\langle cl,\ c,\ bl\rangle,\ a,\ br\rangle \\
\qquad\quad \mathsf{else\ if}\ a < b \\
\qquad\qquad \mathsf{then\ if}\ bl = \langle\rangle\ \mathsf{then}\ \langle\langle cl,\ c,\ bl\rangle,\ b,\ br\rangle \\
\qquad\qquad\quad \mathsf{else\ case}\ splay\ a\ bl\ \mathsf{of} \\
\qquad\qquad\qquad \langle al,\ a',\ ar\rangle \Rightarrow \langle\langle cl,\ c,\ al\rangle,\ a',\ \langle ar,\ b,\ br\rangle\rangle \\
\qquad\qquad \mathsf{else\ if}\ br = \langle\rangle\ \mathsf{then}\ \langle\langle cl,\ c,\ bl\rangle,\ b,\ br\rangle \\
\qquad\qquad\quad \mathsf{else\ case}\ splay\ a\ br\ \mathsf{of} \\
\qquad\qquad\qquad \langle al,\ x,\ xa\rangle \Rightarrow \langle\langle\langle cl,\ c,\ bl\rangle,\ b,\ al\rangle,\ x,\ xa\rangle)
\end{array}
$$

Fig. 1. Function *splay*

$$
\begin{array}{l}
insert_{st}\ a\ t = \\
(\mathit{if}\ t = \langle\rangle\ \mathit{then}\ \langle\langle\rangle,\ a,\ \langle\rangle\rangle \\
\ \mathit{else\ case}\ splay\ a\ t\ \mathit{of} \\
\qquad \langle l,\ a',\ r\rangle \Rightarrow \\
\qquad\ \ \mathit{if}\ a = a'\ \mathit{then}\ \langle l,\ a,\ r\rangle \\
\qquad\ \ \mathit{else\ if}\ a < a'\ \mathit{then}\ \langle l,\ a,\ \langle\langle\rangle,\ a',\ r\rangle\rangle\ \mathit{else}\ \langle\langle l,\ a',\ \langle\rangle\rangle,\ a,\ r\rangle)
\end{array}
$$

$$
\begin{array}{l}
delete_{st}\ a\ t = \\
(\mathit{if}\ t = \langle\rangle\ \mathit{then}\ \langle\rangle \\
\ \mathit{else\ case}\ splay\ a\ t\ \mathit{of} \\
\qquad \langle l,\ a',\ r\rangle \Rightarrow \\
\qquad\ \ \mathit{if}\ a = a' \\
\qquad\ \ \mathit{then\ if}\ l = \langle\rangle\ \mathit{then}\ r\ \mathit{else\ case}\ splay\_max\ l\ \mathit{of}\ \langle l',\ m,\ r'\rangle \Rightarrow \langle l',\ m,\ r\rangle \\
\qquad\ \ \mathit{else}\ \langle l,\ a',\ r\rangle)
\end{array}
$$

Fig. 2. Functions $insert_{st}$ and $delete_{st}$

$$
\begin{array}{l}
splay\_max\ \langle\rangle = \langle\rangle \\
splay\_max\ \langle l,\ b,\ \langle\rangle\rangle = \langle l,\ b,\ \langle\rangle\rangle \\
splay\_max\ \langle l,\ b,\ \langle rl,\ c,\ rr\rangle\rangle = \\
(\mathit{if}\ rr = \langle\rangle\ \mathit{then}\ \langle\langle l,\ b,\ rl\rangle,\ c,\ \langle\rangle\rangle \\
\ \mathit{else\ case}\ splay\_max\ rr\ \mathit{of}\ \langle rrl,\ x,\ xa\rangle \Rightarrow \langle\langle\langle l,\ b,\ rl\rangle,\ c,\ rrl\rangle,\ x,\ xa\rangle)
\end{array}
$$

Fig. 3. Function *splay_max*

$$
\begin{array}{l}
set_{tree}\ \langle\rangle = \emptyset \\
set_{tree}\ \langle l,\ a,\ r\rangle = \{a\} \cup (set_{tree}\ l \cup set_{tree}\ r) \\
bst\ \langle\rangle = True \\
bst\ \langle l,\ a,\ r\rangle = \\
(bst\ l \wedge bst\ r \wedge (\forall x \in set_{tree}\ l.\ x < a)\ \wedge (\forall x \in set_{tree}\ r.\ a < x))
\end{array}
$$

Fig. 4. Functions set_{tree} and *bst*

To delete a, you splay with a and if a ends up at the root, you replace it with the maximal element removed from the left subtree. The latter step is performed by *splay_max* that splays with the maximal element.

6.1 Functional Correctness

So far we had ignored functional correctness but for splay trees we actually need it in the verification of the complexity. To formulate functional correctness we

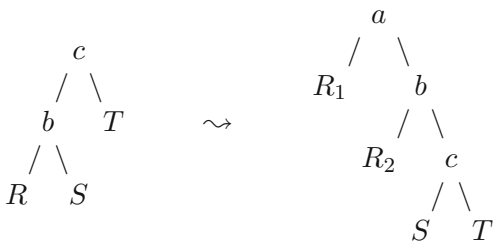

Fig. 5. Zig-zig case for *splay*: $a < b < c$

need the two auxiliary functions shown in Fig. 4. Function set_{tree} collects the elements in the tree, function *bst* checks if the tree is a *binary search tree* according to the linear ordering "<" on the elements. The key functional properties are that splaying does not change the contents of the tree (it merely reorganizes it) and that *bst* is an invariant of splaying:

$$set_{tree}\ (splay\ a\ t) = set_{tree}\ t$$
$$bst\ t \Longrightarrow bst\ (splay\ a\ t)$$

Similar properties can be proved for insertion and deletion, e.g.,

$$bst\ t \Longrightarrow set_{tree}\ (delete_{st}\ a\ t) = set_{tree}\ t - \{a\}$$

Now we present two amortized analyses: a simpler one that yields the bounds proved by Sleator and Tarjan [12] and a more complicated and precise one due to Schoenmakers [11].

6.2 Amortized Analysis

The timing functions are straightforward and not shown. Roughly speaking, they count only the number of splay steps: t_{splay} counts the number of calls of *splay*, t_{splay_max} counts the number of calls of *splay_max*; t_{delete} counts the time for both *splay* and *splay_max*.

The potential of a tree is defined as a sum of logarithms as follows:

$$\varphi\ t = \log_2\ |t|_1$$
$$\Phi\ \langle\rangle = 0$$
$$\Phi\ \langle l,\ a,\ r\rangle = \Phi\ l + \Phi\ r + \varphi\ \langle l,\ a,\ r\rangle$$

The amortized complexity of splaying is defined as usual:

$$\mathcal{A}\ a\ t = t_{splay}\ a\ t + \Phi\ (splay\ a\ t) - \Phi\ t$$

Let *subtrees* yield the set of all subtrees of a tree:

$$subtrees\ \langle\rangle = \{\langle\rangle\}$$
$$subtrees\ \langle l,\ a,\ r\rangle = \{\langle l,\ a,\ r\rangle\} \cup (subtrees\ l \cup subtrees\ r)$$

The following logarithmic bound is proved by induction on t according to the recursion schema of *splay*: if $bst\ t$ and $\langle l,\ a,\ r\rangle \in subtrees\ t$ then

$$\mathcal{A}\ a\ t \le 3 * (\varphi\ t - \varphi\ \langle l,\ a,\ r\rangle) + 1 \tag{3}$$

Let us look at one case of the inductive proof in detail. We pick the so-called zig-zig case shown in Fig. 5. Subtrees with root x are called X on the left and X' on the right-hand side. Thus the figure depicts $splay\ a\ C = A'$ assuming the recursive call $splay\ a\ R = \langle R_1,\ a,\ R_2\rangle =: R'$.

$$\begin{aligned}
&\mathcal{A}\ a\ C = \mathcal{A}\ a\ R + \varphi\ B' + \varphi\ C' - \varphi\ B - \varphi\ R' + 1\\
&\le 3 * (\varphi\ R - \varphi\ \langle l,\ a,\ r\rangle) + \varphi\ B' + \varphi\ C' - \varphi\ B - \varphi\ R' + 2\\
&\quad \text{by ind.hyp.}\\
&= 2 * \varphi\ R + \varphi\ B' + \varphi\ C' - \varphi\ B - 3 * \varphi\ \langle l,\ a,\ r\rangle + 2\\
&\quad \text{because } \varphi\ R = \varphi\ R'\\
&\le \varphi\ R + \varphi\ B' + \varphi\ C' - 3 * \varphi\ \langle l,\ a,\ r\rangle + 2\\
&\quad \text{because } \varphi\ B < \varphi\ R\\
&\le \varphi\ B' + 2 * \varphi\ C - 3 * \varphi\ \langle l,\ a,\ r\rangle + 1\\
&\quad \text{because } 1 + \log_2\ x + \log_2\ y < 2 * \log_2\ (x + y) \text{ if } x, y > 0\\
&\le 3 * (\varphi\ C - \varphi\ \langle l,\ a,\ r\rangle) + 1 \qquad \text{because } \varphi\ B' \le \varphi\ C
\end{aligned}$$

This looks similar to the proof by Sleator and Tarjan but is different: they consider one double rotation whereas we argue about the whole input-output relationship; also our *log* argument is simpler.

From (3) we obtain in the worst case ($l = r = \langle\rangle$):

$$\text{If } bst\ t \text{ and } a \in set_{tree}\ t \text{ then } \mathcal{A}\ a\ t \le 3 * (\varphi\ t - 1) + 1.$$

In the literature the case $a \notin set_{tree}\ t$ is treated informally by stating that it can be reduced to $a' \in set_{tree}\ t$: one could have called *splay* with some $a' \in set_{tree}\ t$ instead of a and the behaviour would have been the same. Formally we prove by induction that if $t \neq \langle\rangle$ and $bst\ t$ then

$$\exists\, a' \in set_{tree}\ t.\ splay\ a'\ t = splay\ a\ t \wedge t_{splay}\ a'\ t = t_{splay}\ a\ t$$

This gives us an upper bound for all binary search trees:

$$bst\ t \Longrightarrow \mathcal{A}\ a\ t \le 3 * \varphi\ t + 1 \tag{4}$$

The $\varphi\ t - 1$ was increased to $\varphi\ t$ because the former is negative if $t = \langle\rangle$.

We also need to determine the amortized complexity $\mathcal{A}m$ of *splay_max*

$$\mathcal{A}m\ t = t_{splay\_max}\ t + \Phi\ (splay\_max\ t) - \Phi\ t$$

A derivation similar to but simpler than the one for $\mathcal{A}$ yields the same upper bound: $bst\ t \Longrightarrow \mathcal{A}m\ t \le 3 * \varphi\ t + 1$.

Now we can apply our amortized analysis theory:

datatype $'a\ op_{st} = Splay\ 'a \mid Insert\ 'a \mid Delete\ 'a$

$nxt_{st}\ (Splay\ a)\ t = splay\ a\ t$ $\quad$ $t_{st}\ (Splay\ a)\ t = t_{splay}\ a\ t$
$nxt_{st}\ (Insert\ a)\ t = insert_{st}\ a\ t$ $\quad$ $t_{st}\ (Insert\ a)\ t = t_{splay}\ a\ t$
$nxt_{st}\ (Delete\ a)\ t = delete_{st}\ a\ t$ $\quad$ $t_{st}\ (Delete\ a)\ t = t_{delete}\ a\ t$

$init = \langle\rangle$ $\quad$ $nxt = nxt_{st}$ $\quad$ $t = t_{st}$ $\quad$ $inv = bst$ $\quad$ $\Phi = \Phi$
$U\ (Splay\ \_)\ t = 3 * \varphi\ t + 1$
$U\ (Insert\ \_)\ t = 4 * \varphi\ t + 2$
$U\ (Delete\ \_)\ t = 6 * \varphi\ t + 2$

The fact that the given U is indeed a correct upper bound follows from the upper bounds for $\mathcal{A}$ and $\mathcal{A}m$; for *Insert* and *Delete* the proof needs more case distinctions and log-manipulations.

6.3 Improved Amortized Analysis

This subsection follows the work of Schoenmakers [11] (except that he confines himself to *splay*) who improves upon the constants in the above analysis. His analysis is parameterized by two constants $\alpha > 1$ and β subject to three constraints where all the variables are assumed to be ≥ 1:

$$(x + y) * (y + z)^\beta \leq (x + y)^\beta * (x + y + z)$$

$$\alpha * (l' + r') * (lr + r)^\beta * (lr + r' + r)^\beta \\ \leq (l' + r')^\beta * (l' + lr + r')^\beta * (l' + lr + r' + r)$$

$$\alpha * (l' + r') * (l' + ll)^\beta * (r' + r)^\beta \\ \leq (l' + r')^\beta * (l' + ll + r')^\beta * (l' + ll + r' + r)$$

The following upper bound is again proved by induction but this time with the help of the above constraints: if $bst\ t$ and $\langle l, a, r\rangle \in subtrees\ t$ then

$$\mathcal{A}\ a\ t \leq \log_\alpha\ (|t|_1\ /\ (|l|_1 + |r|_1)) + 1$$

From this we obtain the following main theorem just like before:

$$\mathcal{A}\ a\ t \leq \log_\alpha\ |t|_1 + 1$$

Now we instantiate the above abstract development with $\alpha = \sqrt[3]{4}$ and $\beta = 1/3$ (which includes proving the three constraints on α and β above) to obtain a bound for splaying that is only half as large as in (4):

$$bst\ t \Longrightarrow \mathcal{A}_{34}\ a\ t \leq 3\ /\ 2 * \varphi\ t$$

The subscript $_{34}$ is our indication that we refer to the $\alpha = \sqrt[3]{4}$ and $\beta = 1/3$ instance. Schoenmakers additionally showed that this specific choice of α and β yields the minimal upper bound.

A similar but simpler development leads to the same bound for $\mathcal{A}m_{34}$ as for $\mathcal{A}_{34}$. Again we apply our amortized analysis theory to verify upper bounds for *Splay*, *Insert* and *Delete* that are also only half as large as before:

$U\ (Splay\ \_)\ t = 3\ /\ 2 * \varphi\ t + 1$
$U\ (Insert\ \_)\ t = 2 * \varphi\ t + 3\ /\ 2$
$U\ (Delete\ \_)\ t = 3 * \varphi\ t + 2$

The proofs in this subsection require a lot of highly nonlinear arithmetic. Only some of the polynomial inequalities can be automated with Harrison's sum-of-squares method [5].

7 Splay Heaps

Splay heaps are another self-adjusting data structure and were invented by Okasaki [10]. Splay heaps are organized internally like splay trees but they implement a priority queue interface. When inserting an element x into a splay heap, the splay heap is first partitioned (by rotations, like *splay*) into two trees, one $\leq x$ and one $> x$, and x becomes the new root:

$$insert\ x\ h = (\mathsf{let}\ (l,\ r) = partition\ x\ h\ \mathsf{in}\ \langle l,\ x,\ r\rangle)$$

$$\begin{array}{l}
partition\ p\ \langle\rangle = (\langle\rangle,\ \langle\rangle)\\
partition\ p\ \langle al,\ a,\ ar\rangle =\\
(\mathsf{if}\ a \leq p\\
\ \mathsf{then\ case}\ ar\ \mathsf{of}\ \langle\rangle \Rightarrow (\langle al,\ a,\ ar\rangle,\ \langle\rangle)\\
\qquad \mid \langle bl,\ b,\ br\rangle \Rightarrow\\
\qquad\quad \mathsf{if}\ b \leq p\ \mathsf{then\ let}\ (pl,\ y) = partition\ p\ br\ \mathsf{in}\ (\langle\langle al,\ a,\ bl\rangle,\ b,\ pl\rangle,\ y)\\
\qquad\quad \mathsf{else\ let}\ (pl,\ pr) = partition\ p\ bl\ \mathsf{in}\ (\langle al,\ a,\ pl\rangle,\ \langle pr,\ b,\ br\rangle)\\
\ \mathsf{else\ case}\ al\ \mathsf{of}\ \langle\rangle \Rightarrow (\langle\rangle,\ \langle al,\ a,\ ar\rangle)\\
\qquad \mid \langle bl,\ b,\ br\rangle \Rightarrow\\
\qquad\quad \mathsf{if}\ b \leq p\ \mathsf{then\ let}\ (pl,\ pr) = partition\ p\ br\ \mathsf{in}\ (\langle bl,\ b,\ pl\rangle,\ \langle pr,\ a,\ ar\rangle)\\
\qquad\quad \mathsf{else\ let}\ (pl,\ pr) = partition\ p\ bl\ \mathsf{in}\ (pl,\ \langle pr,\ b,\ \langle br,\ a,\ ar\rangle\rangle))
\end{array}$$

Function *del_min* removes the minimal element and is similar to *splay_max*:

$$\begin{array}{l}
del_min\ \langle\rangle = \langle\rangle\\
del_min\ \langle\langle\rangle,\ uu,\ r\rangle = r\\
del_min\ \langle\langle ll,\ a,\ lr\rangle,\ b,\ r\rangle =\\
(\mathsf{if}\ ll = \langle\rangle\ \mathsf{then}\ \langle lr,\ b,\ r\rangle\ \mathsf{else}\ \langle del_min\ ll,\ a,\ \langle lr,\ b,\ r\rangle\rangle)
\end{array}$$

In contrast to search trees, priority queues may contain elements multiple times. Therefore splay heaps satisfy the weaker invariant bst_{eq}:

$$\begin{array}{l}
bst_{eq}\ \langle\rangle = True\\
bst_{eq}\ \langle l,\ a,\ r\rangle =\\
(bst_{eq}\ l \wedge bst_{eq}\ r \wedge (\forall x \in set_{tree}\ l.\ x \leq a)\ \wedge (\forall x \in set_{tree}\ r.\ a \leq x))
\end{array}$$

This is an invariant for both *partition* and *del_min*:

If $bst_{eq}\ t$ and $partition\ p\ t = (l,\ r)$ then $bst_{eq}\ \langle l,\ p,\ r\rangle$.
If $bst_{eq}\ t$ then $bst_{eq}\ (del\_min\ t)$.

For the functional correctness proofs see [9].

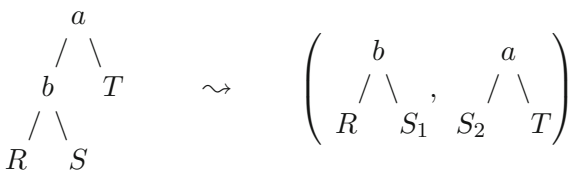

Fig. 6. Zig-zag case for *partition*: $b \leq p < a$

7.1 Amortized Analysis

Now we verify the amortized analysis due to Okasaki. The timing functions are straightforward and not shown: t_{part} and t_{dm} count the number of calls of *partition* and *del_min*. The potential of a tree is defined as for splay trees in Sect. 6.2. The following logarithmic bound of the amortized complexity $\mathcal{A}\ p\ t = t_{part}\ p\ t + \Phi\ l' + \Phi\ r' - \Phi\ t$ is proved by computation induction on *partition* t: if $bst_{eq}\ t$ and *partition* $p\ t = (l', r')$ then

$$\mathcal{A}\ p\ t \leq 2 * \varphi\ t + 1$$

Okasaki [10] shows the zig-zig case of the induction, I show the zig-zag case in Fig. 6. Subtrees with root x are called X on the left and X' on the right-hand side. Thus Fig. 6 depicts *partition* $p\ A = (B', A')$ assuming the recursive call *partition* $p\ S = (S_1, S_2)$.

$$\begin{aligned}
&\mathcal{A}\ p\ A = \mathcal{A}\ p\ S + 1 + \varphi\ B' + \varphi\ A' - \varphi\ B - \varphi\ A \\
&\leq 2 * \varphi\ S + 2 + \varphi\ B' + \varphi\ A' - \varphi\ B - \varphi\ A \qquad \text{by ind.hyp.} \\
&= 2 + \varphi\ B' + \varphi\ A' \qquad \text{because } \varphi\ S < \varphi\ B \text{ and } \varphi\ S < \varphi\ A \\
&\leq 2 * \log_2\ (|R|_1 + |S_1|_1 + |S_2|_1 + |T|_1 - 1) + 1 \\
&\quad \text{because } 1 + \log_2\ x + \log_2\ y \leq 2 * \log_2\ (x + y - 1) \text{ if } x, y \geq 2 \\
&= 2 * \varphi\ A + 1 \qquad \text{because } |S_1| + |S_2| = |S|
\end{aligned}$$

The proof of the amortized complexity of *del_min* is similar to the one for *splay_max*: $t_{dm}\ t + \Phi\ (\mathit{del\_min}\ t) - \Phi\ t \leq 2 * \varphi\ t + 1$. Now it is routine to verify the following amortized complexities by instantiating our standard theory with $U\ (\mathit{Insert}\ \_)\ t = 3 * \log_2\ (|t|_1 + 1) + 1$ and $U\ \mathit{Delmin}\ t = 2 * \varphi\ t + 1$.

Acknowledgement. Berry Schoenmakers patiently answered many questions about his work whenever I needed help.

References

1. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods Comput. Sci. **7**(2), 33 (2011)
2. Charguéraud, A., Pottier, F.: Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In: Urban, C., Zhang, X. (ed.) ITP 2015. LNCS, vol. 9236, pp. 137–154. Springer, Heidelberg (2015)

3. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, New York (1990)
4. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The pairing heap: A new form of self-adjusting heap. Algorithmica **1**(1), 111–129 (1986)
5. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 102–118. Springer, Heidelberg (2007)
6. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. **34**(3), 14 (2012)
7. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: Proceedings of the 30th ACM Symposium Principles of Programming Languages, pp. 185–197 (2003)
8. Kaldewaij, A., Schoenmakers, B.: The derivation of a tighter bound for top-down skew heaps. Inf. Process. Lett. **37**, 265–271 (1991)
9. Nipkow, T.: Amortized complexity verified. Archive of Formal Proofs (2014). http://afp.sf.net/entries/Amortized_Complexity.shtml. Formal proof development
10. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)
11. Schoenmakers, B.: A systematic analysis of splaying. Inf. Process. Lett. **45**, 41–50 (1993)
12. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. ACM **32**(3), 652–686 (1985)
13. Sleator, D.D., Tarjan, R.E.: Self-adjusting heaps. SIAM J. Comput. **15**(1), 52–69 (1986)
14. Tarjan, R.E.: Amortized complexity. SIAM J. Alg. Disc. Meth. **6**(2), 306–318 (1985)
15. Traytel, D., Berghofer, S., Nipkow, T.: Extending hindley-milner type inference with coercive structural subtyping. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 89–104. Springer, Heidelberg (2011)