

Amortized Complexity Verified

Tobias Nipkow
Technische Universität München

Abstract

A framework for the analysis of the amortized complexity of (functional) data structures is formalized in Isabelle/HOL and applied to two non-trivial examples: skew heaps and splay trees. In the same spirit we show that the move-to-front algorithm for the list update problem is 2-competitive.

1 Introduction

Amortized complexity [10, 2] of an algorithm averages the running times of a sequence of invocations of the algorithm. This refined kind of analysis is useful if a single invocation may show some worst-case behaviour that averages out over time. A classical example is the incrementation of a sequence of bits by going through them one by one, starting with the least significant one, until one finds a 0. The worst case of a sequence of 1s is rare, and it results in a new number where the next few incrementations are very fast. It turns out (see Section 3) that the amortized complexity is constant rather than linear. In contrast to worst-case complexity, where one can often argue in terms of some structural property of the data structure (e.g. that certain search trees have a certain maximal height), amortized complexity requires more refined arguments.

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 (including the binary counter mentioned above) and the more challenging examples of skew heaps and especially splay trees. Finally we also verify the complexity of the mother of all online algorithms, the move-to-front algorithm: we show that it is at most twice as slow as the best possible offline algorithm.

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 kind of language the algorithm is implemented. In fact, the standard imperative implementations of all of our examples have the same complexity as the functional model.

2 Basics

We follow the usual Isabelle/HOL syntax except for a few obvious minor deviations, like writing $|_$ for all kinds of size functions, including *card*. Instead of $x \# xs$ we write $x \cdot xs$. Note that Isabelle supports implicit coercions, in particular from *nat* to *real*, that are inserted automatically [11].

We also make use of ordinary binary trees:

```
datatype 'a tree = Tip | Node ('a tree) 'a ('a tree)
|Tip| = 0
|Node t1 a t2| = |t1| + |t2| + 1
```

3 Amortized Analysis Formalized

We formalize the following scenario. We are given a number of operations that may take parameters and that update some state as a side effect. The state is some possibly large data structure. 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.

next :: *'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 $\implies$ inv (next 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. *next f s*.

The effect of each operation *f* is modeled as a function *next f* from state to state. You may immediately wonder how we can talk about the complexity of a function (as opposed to an algorithm). Strictly speaking we cannot, which is why our model has another parameter, namely *t*. Note that *t* need not be a closed form expression for the actual complexity, it can and often will be recursive. More precisely, the definition of *t* will follow the structure

of the definition of nxt precisely. Thus it can be seen easily that t does indeed describe the time nxt (interpreted as an algorithm) takes to execute. Alternatively one can view this as a model of all implementations of nxt that have the given complexity t . 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.

There are various methods to analyze amortized complexity. The most popular one is the *potential method*. This is the one we formalize. That is, our theory has another parameter:

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

You should think of the potential of the state as an indication of how hard to operate on the data structure is: the higher the potential, the harder. 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 :

$$\begin{aligned} state &:: (nat \Rightarrow 'o) \Rightarrow nat \Rightarrow 's \\ state\ f\ 0 &= init \\ state\ f\ (Suc\ n) &= nxt\ (f\ n)\ (state\ f\ n) \end{aligned}$$

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

$$\begin{aligned} a &:: (nat \Rightarrow 'o) \Rightarrow nat \Rightarrow real \\ a\ f\ i &= t\ (f\ i)\ (state\ f\ i) + \Phi\ (state\ f\ (i + 1)) - \Phi\ (state\ f\ i) \end{aligned}$$

If the potential increases, the amortized complexity is higher than the real one. This can be seen as paying some money into an account that finances later operations where the potential decreases. (This view is also known as the *banker's method*.) 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 $inv\ s \implies t\ f\ s + \Phi\ (nxt\ f\ s) - \Phi\ s \leq U\ f\ s$

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

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

4 Easy Examples

Unless noted otherwise, the examples in this section come from a standard textbook [2].

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:

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

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

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

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

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

The key idea of the analysis is to define the potential p as the number of *True* bits in the list. 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

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

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

$$\begin{aligned} \text{next}_{stk} (\text{Push } x) \text{ } xs &= x \cdot xs \\ \text{next}_{stk} (\text{Pop } n) \text{ } xs &= \text{drop } n \text{ } xs \end{aligned}$$

In complete analogy the running time function is defined:

$$\begin{aligned} t_{stk} (\text{Push } x) \text{ } xs &= 1 \\ t_{stk} (\text{Pop } n) \text{ } xs &= \min n \mid xs \end{aligned}$$

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

$$\begin{aligned} \text{init} &= [] & \text{next} &= \text{next}_{stk} & t &= t_{stk} & \text{inv } s &= \text{True} \\ \Phi &= \text{length} & U f s &= (\text{case } f \text{ of } \text{Push } x \Rightarrow 2 \mid \text{Pop } n \Rightarrow 0) \end{aligned}$$

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. Just for a change we ignore the actual elements because they are irrelevant for the complexity analysis. Therefore the operations

$$\text{datatype } \text{op}_{tb} = \text{Ins} \mid \text{Del}$$

do not have arguments. Similarly we only record the size of the table at any point. More precisely the state is a pair of natural numbers (n, l) that abstracts a table of size l with n elements. This is how the operations behave:

$$\begin{aligned} \text{next}_{tb} \text{ Ins } (n, l) &= (n + 1, \text{if } n < l \text{ then } l \text{ else if } l = 0 \text{ then } 1 \text{ else } 2 * l) \\ \text{next}_{tb} \text{ Del } (n, l) &= \\ (n - 1, \text{if } n = 1 \text{ then } 0 \text{ else if } 4 * (n - 1) < l \text{ then } l \text{ div } 2 \text{ else } l) \end{aligned}$$

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:

$$\begin{aligned} t_{tb} \text{ Ins } (n, l) &= (\text{if } n < l \text{ then } 1 \text{ else } n + 1) \\ t_{tb} \text{ Del } (n, l) &= (\text{if } n = 1 \text{ then } 1 \text{ else if } 4 * (n - 1) < l \text{ then } n \text{ else } 1) \end{aligned}$$

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:

$$\begin{aligned} \text{init} &= (0, 0) & \text{next} &= \text{next}_{tb} & t &= t_{tb} \\ \text{inv } (n, l) &= (\text{if } l = 0 \text{ then } n = 0 \text{ else } n \leq l \wedge l \leq 4 * n) \end{aligned}$$

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

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

Now it is automatic to show that the amortized complexity of *Ins* and *Del* is bounded by 3 and 2:

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

4.4 Queues

Queues have one operations for enqueueing an item and one for dequeuing the oldest item:

$$\text{datatype } 'a \text{ op}_q = Enq \ 'a \mid Del$$

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

The simplest possible implementation of functional queues consists of two lists (stacks) (xs, ys) (e.g. [4]):

$$\begin{aligned} \text{next}_q (Enq \ x) (xs, ys) &= (x \cdot xs, ys) \\ \text{next}_q Del (xs, ys) &= (\text{if } ys = [] \text{ then } ([], tl (rev \ xs)) \text{ else } (xs, tl \ ys)) \\ t_q (Enq \ x) (xs, ys) &= 1 \\ t_q Del (xs, ys) &= (\text{if } ys = [] \text{ then } |xs| \text{ else } 0) \end{aligned}$$

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

$$\begin{array}{lll} \text{init} = ([], []) & \text{next} = \text{next}_q & t = t_q \\ \text{inv } s = \text{True} & \Phi(xs, ys) = |xs| & Uf s = 2 \end{array}$$

5 Skew Heaps

This section analyzes a beautifully simple data structure for priority queues: skew heaps [9]. 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:

$$\begin{aligned} \text{meld } h_1 \ h_2 = & \\ (\text{case } h_1 \text{ of } Tip \Rightarrow h_2 & \\ \mid Node \ l_1 \ a_1 \ r_1 \Rightarrow & \\ \text{case } h_2 \text{ of } Tip \Rightarrow h_1 & \end{aligned}$$

$$\begin{aligned}
& | \text{Node } l_2 \ a_2 \ r_2 \Rightarrow \\
& \quad \text{if } a_1 \leq a_2 \text{ then } \text{Node } (\text{meld } h_2 \ r_1) \ a_1 \ l_1 \\
& \quad \text{else } \text{Node } (\text{meld } h_1 \ r_2) \ a_2 \ l_2
\end{aligned}$$

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

$$\text{datatype } 'a \text{ op}_{pq} = \text{Insert } 'a \mid \text{Delmin}$$

They are implemented via *meld* as follows:

$$\begin{aligned}
\text{next}_{pq} (\text{Insert } a) \ h &= \text{meld } (\text{Node } \text{Tip } a \ \text{Tip}) \ h \\
\text{next}_{pq} \ \text{Delmin} \ h &= (\text{case } h \text{ of } \text{Tip} \Rightarrow \text{Tip} \mid \text{Node } t_1 \ a \ x \Rightarrow \text{meld } t_1 \ x)
\end{aligned}$$

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 [3] that we will follow. Their cost measure counts the number of calls of *meld*, *Insert* and *Delmin*:

$$\begin{aligned}
t_{\text{meld}} \ \text{Tip} \ h &= 1 \\
t_{\text{meld}} \ h \ \text{Tip} &= 1 \\
t_{\text{meld}} \ (\text{Node } l_1 \ a_1 \ r_1) \ (\text{Node } l_2 \ a_2 \ r_2) &= \\
& (\text{if } a_1 \leq a_2 \text{ then } t_{\text{meld}} \ (\text{Node } l_2 \ a_2 \ r_2) \ r_1 \text{ else } t_{\text{meld}} \ (\text{Node } l_1 \ a_1 \ r_1) \ r_2) + 1 \\
t_{pq} (\text{Insert } a) \ h &= t_{\text{meld}} \ (\text{Node } \text{Tip } a \ \text{Tip}) \ h + 1 \\
t_{pq} \ \text{Delmin} \ h &= (\text{case } h \text{ of } \text{Tip} \Rightarrow 1 \mid \text{Node } t_1 \ a \ t_2 \Rightarrow t_{\text{meld}} \ t_1 \ t_2 + 1)
\end{aligned}$$

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:

$$\begin{aligned}
\Phi \ \text{Tip} &= 0 \\
\Phi \ (\text{Node } l \ _r) &= \Phi \ l + \Phi \ r + (\text{if } |l| < |r| \text{ then } 1 \text{ else } 0)
\end{aligned}$$

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

$$\begin{aligned}
\text{rheavy} \ (\text{Node } l \ _r) &= (|l| < |r|) \\
\text{lpath} \ \text{Tip} &= [] \\
\text{lpath} \ (\text{Node } l \ a \ r) &= \text{Node } l \ a \ r \cdot \text{lpath } l \\
\text{rpath} \ \text{Tip} &= [] \\
\text{rpath} \ (\text{Node } l \ a \ r) &= \text{Node } l \ a \ r \cdot \text{rpath } r \\
\Gamma \ h &= |\text{filter } \text{rheavy} \ (\text{lpath } h)| \\
\Delta \ h &= |\text{filter } (\lambda p. \neg \text{rheavy } p) \ (\text{rpath } h)|
\end{aligned}$$

Two easy inductive properties:

$$2^{\Gamma h} \leq |h| + 1 \quad (1) \quad 2^{\Delta h} \leq |h| + 1 \quad (2)$$

Now the desired logarithmic amortized complexity of *meld* follows because $\log_b x + \log_b y \leq 2 * \log_b (x + y)$ (if $1 \leq x$, $1 \leq y$ and $1 < b$):

$$\begin{aligned} & t_{\text{meld } t_1 \ t_2} + \Phi(\text{meld } t_1 \ t_2) - \Phi t_1 - \Phi t_2 \\ & \leq \Gamma(\text{meld } t_1 \ t_2) + \Delta t_1 + \Delta t_2 + 1 \quad (\text{by induction on } \text{meld}) \\ & \leq \log_2 (|\text{meld } t_1 \ t_2| + 1) + \log_2 (|t_1| + 1) + \log_2 (|t_2| + 1) + 1 \\ & \quad (\text{by (1), (2)}) \\ & = \log_2 (|t_1| + |t_2| + 1) + \log_2 (|t_1| + 1) + \log_2 (|t_2| + 1) + 1 \\ & \leq \log_2 (|t_1| + |t_2| + 2) + \log_2 (|t_1| + 1) + \log_2 (|t_2| + 1) + 1 \\ & \leq \log_2 (|t_1| + |t_2| + 2) + 2 * \log_2 (|t_1| + 1 + |t_2| + 1) + 1 \\ & = 3 * \log_2 (|t_1| + |t_2| + 2) + 1 \end{aligned}$$

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

$$\begin{aligned} U(\text{Insert } _) h &= 3 * \log_2 (|h| + 3) + 2 \\ U(\text{Delmin}) &= 3 * \log_2 (|h| + 3) + 4 \end{aligned}$$

6 Splay Trees

A splay tree [8] is another self-adjusting data structure, but more subtle than skew heaps. It achieves its amortized logarithmic complexity by local rotations of subtrees along the access path that improve the balancedness of the tree. 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. If the given element is not in the tree, the last element found before a *Tip* was met is rotated to the root. The complete definition is shown in Figure 1.

We consider only two operations: splaying itself and insert:

$$\begin{aligned} & \text{insert}_{st} a \ t = \\ & (\text{if } t = \text{Tip} \text{ then } \text{Node } \text{Tip } a \ \text{Tip} \\ & \quad \text{else case splay } a \ t \text{ of} \\ & \quad \quad \text{Node } l \ e \ r \Rightarrow \\ & \quad \quad \quad \text{if } a = e \text{ then } \text{Node } l \ e \ r \\ & \quad \quad \quad \text{else if } a < e \text{ then } \text{Node } l \ a \ (\text{Node } \text{Tip } e \ r) \\ & \quad \quad \quad \text{else } \text{Node } (\text{Node } l \ e \ \text{Tip}) \ a \ r) \end{aligned}$$

Given splaying, searching for an element in the tree is a trivial: you splay with the given element and check if it ends up at the root afterwards.

```

splay a Tip = Tip
splay a (Node l b r) =
(if a = b then Node l b r
 else if a < b
   then case l of Tip  $\Rightarrow$  Node l b r
   | Node ll c lr  $\Rightarrow$ 
     if a = c then Node ll a (Node lr b r)
     else if a < c
       then if ll = Tip then Node ll c (Node lr b r)
       else case splay a ll of
         Node lll u llr  $\Rightarrow$ 
           Node lll u (Node llr c (Node lr b r))
       else if lr = Tip then Node ll c (Node lr b r)
       else case splay a lr of
         Node lrl u lrr  $\Rightarrow$ 
           Node (Node ll c lrl) u (Node lrr b r)
     else case r of Tip  $\Rightarrow$  Node l b r
     | Node rl c rr  $\Rightarrow$ 
       if a = c then Node (Node l b rl) a rr
       else if a < c
         then if rl = Tip then Node (Node l b rl) c rr
         else case splay a rl of
           Node rll u rlr  $\Rightarrow$ 
             Node (Node l b rll) u (Node rlr c rr)
         else if rr = Tip then Node (Node l b rl) c rr
         else case splay a rr of
           Node rrl x xa  $\Rightarrow$ 
             Node (Node (Node l b rl) c rrl) x xa)

```

Figure 1: Function *splay*

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 need a few auxiliary functions:

$$\begin{aligned}
set_{tree} \text{ Tip} &= \emptyset \\
set_{tree} (\text{Node } l \ x \ r) &= \{x\} \cup (set_{tree} \ l \cup set_{tree} \ r) \\
bst \ \text{Tip} &= \text{True} \\
bst (\text{Node } l \ a \ r) &= \\
& (bst \ l \wedge bst \ r \wedge (\forall x \in set_tree \ l. x < a) \wedge (\forall x \in set_tree \ r. a < x))
\end{aligned}$$

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

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

and that bst is an invariant of splaying:

$$bst \ t \implies bst \ (\text{splay } a \ t)$$

6.2 Amortized Analysis

This subsection follows the work of Schoenmakers [6] who again provides precise definitions and proofs. Warning: his size function for trees starts at 1 for the empty tree, we start at 0.

His analysis is parameterized by two constants $\alpha > 1$ and β subject to three constraints (which are of course derived from the proof):

$$(2 + x + y) * (2 + y + z)^\beta \leq (2 + x + y)^\beta * (3 + x + y + z)$$

$$\begin{aligned}
& \alpha * (2 + l' + r') * (2 + lr + r)^\beta * (3 + lr + r' + r)^\beta \\
& \leq (2 + l' + r')^\beta * (3 + l' + lr + r')^\beta * (l' + lr + r' + r + 4)
\end{aligned}$$

$$\begin{aligned}
& \alpha * (2 + l' + r') * (2 + l' + ll)^\beta * (2 + r' + r)^\beta \\
& \leq (2 + l' + r')^\beta * (3 + l' + ll + r')^\beta * (l' + ll + r' + r + 4)
\end{aligned}$$

where all the variables are assumed to be ≥ 0 . The constraints in [6] look different but are equivalent.

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

$$\varphi \ t_1 \ t_2 = \beta * \log_\alpha (|t_1| + |t_2| + 2)$$

$$\Phi \ \text{Tip} = 0$$

$$\Phi (\text{Node } l \ _ \ r) = \Phi \ l + \Phi \ r + \varphi \ l \ r$$

```

 $t_{splay} \ a \ Tip = 0$ 
 $t_{splay} \ a \ (Node \ l \ b \ r) =$ 
  (if  $a = b$  then 0
   else if  $a < b$ 
     then case  $l$  of  $Tip \Rightarrow 0$ 
       |  $Node \ ll \ c \ lr \Rightarrow$ 
         if  $a = c$  then 0
         else if  $a < c$  then if  $ll = Tip$  then 0 else  $t_{splay} \ a \ ll + 1$ 
         else if  $lr = Tip$  then 0 else  $t_{splay} \ a \ lr + 1$ 
     else case  $r$  of  $Tip \Rightarrow 0$ 
       |  $Node \ rl \ c \ rr \Rightarrow$ 
         if  $a = c$  then 0
         else if  $a < c$  then if  $rl = Tip$  then 0 else  $t_{splay} \ a \ rl + 1$ 
         else if  $rr = Tip$  then 0 else  $t_{splay} \ a \ rr + 1$ )

```

Figure 2: Function t_{splay}

The amortized complexity of splaying is defined as usual:

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

where t_{splay} is defined in Figure 2. We follow Schoenmakers in counting only the number of recursive calls of $splay$.

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

$$\begin{aligned}
subtrees \ Tip &= \{Tip\} \\
subtrees \ (Node \ l \ a \ r) &= \{Node \ l \ a \ r\} \cup (subtrees \ l \cup subtrees \ r)
\end{aligned}$$

The following key upper bound theorem is proved by induction with the help of the constraints assumed at the beginning of Section 6.2:

$$\begin{aligned}
&\llbracket bst \ t; \ Node \ la \ a \ ra \in subtrees \ t \rrbracket \\
&\implies A \ a \ t \leq \log_{\alpha} ((|t| + 1) / (|la| + |ra| + 2))
\end{aligned}$$

In the worst case la and ra can be Tip and we obtain

$$\llbracket bst \ t; \ a \in set_{tree} \ t \rrbracket \implies A \ a \ t \leq \log_{\alpha} ((|t| + 1) / 2)$$

Schoenmakers does not treat the case $a \notin set_{tree} \ t$ but 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. By induction we prove:

$$\begin{aligned}
&\llbracket t \neq Tip; \ bst \ t \rrbracket \\
&\implies \exists a' \in set_{tree} \ t. \ splay \ a' \ t = splay \ a \ t \wedge t_{splay} \ a' \ t = t_{splay} \ a \ t
\end{aligned}$$

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

$$bst\ t \implies A\ a\ t \leq \log_\alpha (|t| + 1)$$

Division by 2 was sacrificed because $\log_\alpha ((|t| + 1) / 2)$ can become negative if $t = Tip$.

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 (after some more formula manipulations) the logarithmic bound for splaying:

$$bst\ t \implies A_{34}\ a\ t \leq 3 / 2 * \log_2 (|t| + 1) \quad (3)$$

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.

Now we apply our amortized analysis theory to a sequence of insertion and splaying operations:

$$\begin{aligned} \text{datatype } 'a\ op_{st} &= Insert\ 'a \mid Splay\ 'a \\ next_{st}\ (Insert\ a)\ t &= insert_{st}\ a\ t \\ next_{st}\ (Splay\ a)\ t &= splay\ a\ t \\ t_{st}\ (Insert\ a)\ t &= t_{splay}\ a\ t \\ t_{st}\ (Splay\ a)\ t &= t_{splay}\ a\ t \\ init &= Tip \quad next = next_{st} \quad t = t_{st} \quad inv = bst \quad \Phi = \Phi_{34} \\ U\ (Insert\ _) \ t &= 2 * \log_2 (|t| + 1) + 1 / 2 \\ U\ (Splay\ _) \ t &= 3 / 2 * \log_2 (|t| + 1) \end{aligned}$$

The fact that the given U is indeed a correct upper bound follows easily from (3) for operation $Splay$; for $Insert$ the proof needs some more $\log$ manipulations.

7 List Update

The list update problem is a simple model to study the competitive analysis of online algorithms (where a sequence of queries arrives one by one) compared to offline algorithms (where the whole sequence of queries is known upfront). In the simplest form of the problem we are given a list of elements that is accessed sequentially from the front and each query asks if some element is in that list. In addition to answering the query the algorithm can rearrange the list by swapping any number of adjacent elements to improve the response time for the next queries. One is usually not interested in the offline algorithm (the offline problem is NP-complete [1]) but merely uses it as a benchmark to compare online algorithms against. We treat the list update problem because it uses the potential method from amortized analysis although we cannot reuse our amortized analysis theory directly.

The archetypal online algorithm is the *move to front* (MTF) heuristic: every time an element is queried, it is moved to the front of the list. This is another instance of a self-organizing data structure, like splay trees. We verify Sleator and Tarjan's result [7] that MTF is 2-competitive, i.e. at most twice as slow as any online algorithm. The proof of competitiveness will compare two lists, that of the online and that of the offline algorithm. When comparing these lists the notion of inversions is crucial.

7.1 Inversions

Given two lists xs and ys , we call a pair (x, y) an *inversion* if x occurs before y in xs but y occurs before x in ys . This requires the notion of the index of an element in a list:

$$\begin{aligned} \text{index } [] \ a &= 0 \\ \text{index } (x \cdot xs) \ a &= (\text{if } x = a \text{ then } 0 \text{ else } \text{index } xs \ a + 1) \end{aligned}$$

Note that $(\text{index } xs \ x = |xs|) = (x \notin \text{set } xs)$. Then we can define what it means that one element occurs before another. We introduce the syntactic sugar " $x < y$ in xs " for readability:

$$x < y \text{ in } xs \iff \text{index } xs \ x < \text{index } xs \ y \wedge y \in \text{set } xs$$

Note that the condition $x \in \text{set } xs$ is implied by the right-hand-side: $y \in \text{set } xs$ implies $\text{index } xs \ y < |xs|$ and hence $\text{index } xs \ x < \text{index } xs \ y \wedge y \in \text{set } xs$ implies $\text{index } xs \ x < |xs|$, which in turn implies $x \in \text{set } xs$.

The set of inversions is defined like this:

$$\text{Inv } xs \ ys = \{(x, y) \mid x < y \text{ in } xs \wedge y < x \text{ in } ys\}$$

The notion of inversion is usually defined for permutations. Similarly we will often need that the two lists are lists of distinct elements and permutations of each other:

$$\text{dist_perm } xs \ ys \equiv \text{distinct } xs \wedge \text{distinct } ys \wedge \text{set } xs = \text{set } ys$$

This implies $|xs| = |ys|$ and is clearly an equivalence.

The measure $|\text{Inv } xs \ ys|$ is a metric: $\text{Inv } xs \ xs = \emptyset$, $|\text{Inv } xs \ ys| = |\text{Inv } ys \ xs|$ and

$$\begin{aligned} &[\text{dist_perm } xs \ ys; \text{dist_perm } ys \ zs] \\ \implies &|\text{Inv } xs \ zs| \leq |\text{Inv } xs \ ys| + |\text{Inv } ys \ zs| \end{aligned}$$

7.2 Move to Front

Function *mtf* moves an element to the front of a list, if it is in the list:

$$\begin{aligned} \text{mtf } x \ xs &= \\ &(\text{if } x \in \text{set } xs \\ &\text{then } x \cdot \text{take } (\text{index } xs \ x) \ xs \ @ \ \text{drop } (\text{index } xs \ x + 1) \ xs \text{ else } xs) \end{aligned}$$

The second important operation is that of swapping two adjacent elements in a list:

$$\text{swapSuc } n \text{ } xs = xs[n := xs ! \text{Suc } n, \text{Suc } n := xs ! n]$$

This is defined with the help of two auxiliary functions: $xs[n := x]$ updates list xs at index n with x and $xs ! n$ is the n -th element of xs . The index of the head of a list is 0.

Let us be more precise about the algorithms that we want to model:

- The state of the algorithm is a list xs of distinct elements.
- Each query consists of an element x . The algorithm performs a linear search for x in xs . This takes time $\text{index } xs \ x + 1$.

After the search:

- The online algorithm MTF moves x to the front, if $x \in \text{set } xs$. This is for free.
- The offline algorithm performs an arbitrary number of swaps of adjacent elements. Each swap takes one time unit. This models the fact that the offline algorithm knows the future and can prepare for it.

The original paper by Sleator and Tarjan [7] considered two kinds of swaps, free and paid ones, and the claim (Theorem 3) that the offline algorithm does not need paid swaps. Later Reingold and Westbrook [5] proved that the opposite is true: the offline algorithm needs only paid swaps. This may also be considered as an argument in favour of verification.

Now we analyze the running time of MTF in comparison to an arbitrary offline algorithm for some arbitrary initial state and arbitrary sequence of queries. That is, we assume the following parameters:

$\text{init} :: 'a \text{ list}$ is the initial state. We assume *distinct init*.

$q :: \text{nat} \Rightarrow 'a$ is an infinite sequence of queries.

$\text{sw}_{\text{off}} :: \text{nat} \Rightarrow \text{nat list}$ models the swaps the offline algorithm performs: $\text{sw}_{\text{off}} \ n$ is the list of swaps performed after the n -th query. Performing the swaps on a state xs is expressed as $\text{foldr } \text{swapSuc } (\text{sw}_{\text{off}} \ n) \ xs$. We assume that each swap is “in range”: $\forall i \in \text{set } (\text{sw}_{\text{off}} \ n). i + 1 < |\text{init}|$. Note that the length of the state does not change over time, hence $|\text{init}|$ is fine here.

We can now define the state (s) and the running time (t) of MTF and the offline algorithm at time n :

$$\begin{aligned}
s_{mtf} \ 0 &= init \\
s_{mtf} \ (Suc \ n) &= mtf \ (q \ n) \ (s_{mtf} \ n) \\
s_{off} \ 0 &= init \\
s_{off} \ (Suc \ n) &= foldr \ swapSuc \ (sw_{off} \ n) \ (s_{off} \ n) \\
t_{mtf} \ n &= index \ (s_{mtf} \ n) \ (q \ n) + 1 \\
t_{off} \ n &= index \ (s_{off} \ n) \ (q \ n) + 1 + |sw_{off} \ n|
\end{aligned}$$

Note that the running times t_{mtf} and t_{off} are not those of the corresponding functional programs but that they are fixed in the literature [7]. That is, we are clearly verifying some abstract model here, not some concrete algorithm.

If we define the potential as the number of inversions that separates MTF from the offline algorithm ($\Phi \ n = |Inv \ (s_{off} \ n) \ (s_{mtf} \ n)|$) we can prove 2-competitiveness of MTF:

$$\begin{aligned}
t_{mtf} \ n + \Phi \ (Suc \ n) - \Phi \ n &\leq 2 * t_{off} \ n \\
(\sum_{i < n} t_{mtf} \ i) &\leq 2 * (\sum_{i < n} t_{off} \ i)
\end{aligned}$$

The proof of the first proposition is a little bit tricky and requires a number of lemmas about inversions that formalize what is often given as a pictorial argument. The second proposition is an easy consequence.

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

References

- [1] C. Ambühl. Offline list update is NP-hard. In M. Paterson, editor, *Algorithms - ESA 2000*, volume 1879 of *Lect. Notes in Comp. Sci.*, pages 42–51. Springer-Verlag, 2000.
- [2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [3] A. Kaldewaij and B. Schoenmakers. The derivation of a tighter bound for top-down skew heaps. *Information Processing Letters*, 37:265–271, 1991.
- [4] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [5] N. Reingold and J. Westbrook. Optimum off-line algorithms for the list update problem. Technical Report 805, Yale University, Department of Computer Science, 1990.

- [6] B. Schoenmakers. A systematic analysis of splaying. *Information Processing Letters*, 45:41–50, 1993.
- [7] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.
- [8] D. D. Sleator and R. E. Tarjan. Self-adjusting bairnary search trees. *J. ACM*, 32(3):652–686, 1985.
- [9] D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, 1986.
- [10] R. E. Tarjan. Amortized complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, 1985.
- [11] D. Traytel, S. Berghofer, and T. Nipkow. Extending Hindley-Milner type inference with coercive structural subtyping. In H. Yang, editor, *Programming Languages and Systems (APLAS 2011)*, volume 7078 of *Lect. Notes in Comp. Sci.*, pages 89–104. Springer-Verlag, 2011.