J. Automated Reasoning manuscript No.
(will be inserted by the editor)

Amortized Complexity Verified

Tobias Nipkow and Hauke Brinkop

Abstract A framework for the analysis of the amortized complexity of functional data structures is formalized in the proof assistant Isabelle/HOL and applied to a number of standard examples and to the following non-trivial ones: skew heaps, splay trees, splay heaps and pairing heaps. The proofs are completely algebraic and are presented in some detail.

1 Introduction

Amortized analysis [43, 7] bounds the average cost of an operation in a sequence of operations in the worst case. In this paper we formalize a simple framework for amortized analysis of functional programs in the theorem prover Isabelle/HOL [37] and apply it to both the easy standard examples and the more challenging examples of skew heaps, splay trees, splay heaps and pairing heaps. This is an extended version of a previous publication [32]: the framework is generalized from unary to *n*-ary operations and an analysis of pairing heaps has been added.

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.

The contributions of the paper are as follows:

- A lightweight and flexible framework for amortized complexity analysis in a theorem prover.
- The first complexity analyses for skew heaps, splay trees, splay heaps and pairing heaps in a theorem prover.
- The first purely algebraic proof of the amortized complexity of pairing heaps.

The last point needs some explanation. For pairing heaps we had to recast the original proof by Fredman *et al.* [13] into algebraic form where intuitive arguments about

Supported by DFG Koselleck grant NI 491/16-1

Technische Universität München

programs and algebraic proofs about numbers are fused into one. For the other non-trivial data structures treated in our paper such proofs could already be found in the literature.

All lemmas and proofs in this paper have been verified with Isabelle/HOL and are available online [29]. The proofs from Sections 3 and 4 are simple and automatic. Many other proofs are more complicated and require longer proof texts. Fortunately Isabelle's proof language Isar [47,36] supports readable proofs, in particular chains of (in)equations. Therefore any such proof that you see in this paper is a faithful rendering of the original Isabelle proof.

It should be noted that this article is not meant to supersede the existing literature on splay trees etc., but to complement the intuition and the pictures already found in the literature with precise notions, code and proofs not yet found in the literature.

1.1 Functions, Code, Time and Trust

We express all algorithms as functions in the logic. Because mathematical functions do not have a complexity they need to be accompanied by definitions of the intended cost functions. Where this cost function comes from is orthogonal to our work. In this paper the cost functions are defined separately but follow the recursive structure of the actual function definitions. Thus the cost function can be seen as an abstract interpretation of the value function. The user is free to choose the granularity of the abstraction and hence of the complexity analysis. In our examples we typically count (recursive) function calls, which is similar to counting loop iterations. Of course separate user-defined cost functions need to be trusted. This is not an issue for our small examples where the correspondence is easily checked by inspection but it becomes one for larger amounts of code. There are two standard solutions. One can automate a translation/abstraction from the definition of an operation to the definition of its cost function. This translation becomes part of the trusted code basis. Alternatively one can define the operations in a monadic style where the value and the cost are computed simultaneously, but the cost is hidden. In a second step one can then project the result of the monadic computation onto the value and the cost, which may also involve trusted code. We have detailed a monadic approach for HOL elsewhere [35]. In the end, the actual means for defining the cost function are orthogonal to the aim of this paper, namely *amortized analysis*.

Isabelle/HOL can generate code in Standard ML, OCaml, Haskell and Scala from function definitions in the logic [15]. Therefore Isabelle/HOL is a high level programming language and we verify code in that language. Of course at this point the compiler chain also becomes part of the trusted code basis. We have to trust the translation from Isabelle/HOL to Standard ML and the Standard ML compiler w.r.t. functional correctness and preservation of asymptotic upper complexity bounds. There is a verified (w.r.t. functional correctness) compilation chain from Isabelle/HOL via CakeML to machine code [21,25].

1.2 Related Work

References to the algorithms literature are spread throughout the paper. Here we concentrate on formal approaches to resource analysis, a rich area that we can only skim.

Early work on automatic complexity analysis includes Wegbreit's METRIC system [46] for LISP, Le Métayer's ACE system [26] for FP, and Benzinger's ACA system [2] for NUPRL. Hickey and Cohen [17] and Flajolet *et al.* [12] focussed on average-case complexity. The rest of the work we cite (with the exception of [46]) deals with worst-case complexity, possibly amortized.

Because of our simple examples (first-order functions, eager evaluation) it is easy to derive the timing functions for our operations by hand (or automatically). Sands [39] has shown how to infer cost functions for a lazy higher-order functional language. Vasconcelos and Hammond [45] describe type-based inference of cost functions for a strict higher-order functional language.

Rather than inferring and analyzing resource consumption externally one can also use the programming language itself for that purpose. Type systems are a popular framework for tracking resources, e.g. [8,9,28]. The last two papers follow the same monadic, dependently typed approach in different theorem provers. Atkey [1] goes one step further. He formalizes a separation logic that supports amortized resource analysis for an imperative language in Coq and proves the logic correct w.r.t. a semantics. Danner *et al.* [11] verify a cost analysis for a functional language; this work was later generalized from lists to inductive types [10].

Hofmann and Jost [20] pioneered automatic type-based amortized analysis of heap usage of functional programs. Particularly impressive are the later generalizations by Hoffmann *et al.* (e.g. [18,19], although currently restricted to polynomials). Carbonneaux *et al.* [4] have implemented a system that goes one step further in that it can generate Coq proof objects that certify the inferred polynomial resource bounds. Madhavan *et al.* [27] can deal with many challenging lazy algorithms.

In summary one can say that there is a whole spectrum of approaches that differ in expressive power, in the complexity of the examples that have been dealt with, and in automation. Of the papers above, only [28,27] contain examples involving logarithms (instead of merely polynomials) and they are simpler than the ones in this paper and not amortized. Like our paper, the work of Charguéraud and Pottier [6,5] is at the complex, interactive end: they verify the almost-linear amortized complexity of an imperative Union-Find implementation in OCaml in Coq using a separation logic with time credits. This is very impressive because of the challenging complexity argument and because the algorithm is imperative. Their approach is complementary to ours: They start with an OCaml program, translate it automatically into a *characteristic formula* that expresses the behaviour of the program (including time consumption) and add this as an axiom to a Coq theory. We start with a program expressed in Isabelle/HOL and can translate it into multiple target languages.

2 Basic Notation: Lists, Trees and Function Definitions

Lists (type $'a\ list$) are constructed from the empty list $[]$ via the infix cons-operator "$\cdot$", $|xs|$ is the length of xs, set converts a list into the set of its elements.

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 $|t|_1 = |t| + 1$.

Function set_tree returns the set of elements in a tree:

$$set\_tree\ \langle\rangle = \emptyset$$
$$set\_tree\ \langle l, a, r\rangle = set\_tree\ l \cup \{a\} \cup set\_tree\ r$$

Function bst_wrt checks if a tree is a *binary search tree* w.r.t. a relation P on the elements and bst is a specialization to a linear ordering "$<$" on the elements:

$$bst\_wrt\ P\ \langle\rangle = True$$
$$bst\_wrt\ P\ \langle l, a, r\rangle =$$
$$(bst\_wrt\ P\ l \wedge bst\_wrt\ P\ r \wedge (\forall x{\in}set\_tree\ l.\ P\ x\ a) \wedge (\forall x{\in}set\_tree\ r.\ P\ a\ x))$$
$$bst = bst\_wrt\ (<)$$

In this paper we assume that all trees are over linearly ordered element types.

Function definitions with overlapping patterns follow the first-match rule: a rule matches only if the previous rules do not match.

3 Amortized Analysis Formalized

We formalize a data type (data structure) and its associated set of operations as follows. Each operation combines n instances of the data type and may take additional parameters of other types. The n input data structures are combined into a new output data structure. Our model is purely functional, there is no mutation. The number n is the *arity* of the operation. In practice, most operations act on a single data structure. However, there are exceptions, for example union operations on data structures.

Our model of amortized analysis is a theory that is parameterized as follows:

$'s$	is the type of the data structure
$'op$	is the type of operation symbols
$arity :: 'op \to nat$	is the arity of each operation
$exec :: 'op \to 's\ list \to 's$	executes an operation
$inv :: 's \to bool$	is an invariant
$cost :: 'op \to 's\ list \to nat$	is the cost function

This theory comes with the assumption that inv is preserved by all operations:

$$(\forall s{\in}set\ ss.\ inv\ s) \wedge |ss| = arity\ f \Longrightarrow inv\ (exec\ f\ ss)$$

This means that any instantiation of this theory also requires the user to prove this proposition.

Type $'op$ will always be instantiated by an enumeration type representing the signature, i.e. the set of operation symbols. Function *exec* maps operation symbols to their meaning, which is a function from a list of data structures to a data structure. If an operation symbol also takes arguments of types other than $'s$, then the operation symbol becomes a function. For example, the stack operation symbol *Push* will have type $'a \to 'op$ where $'a$ is the type of stack elements. Thus pushing 1 onto a stack s is denoted by $exec\ (Push\ 1)\ [s]$. Since functions are extensional, the execution cost of an operation is modeled explicitly: $cost\ f\ ss$ is the cost of running $exec\ f\ ss$. The relationship between *exec* and *cost* has been discussed in Section 1.1. In particular, *cost* need not be a closed form expression for the actual complexity. In the rest of the paper, unless stated otherwise, cost is synonymous with time and is measured in terms of the number of function calls.

We formalize the *potential method* for amortized analysis. That is, our theory has another parameter, the *potential*:

$$\Phi :: 's \to real$$

We assume it never becomes negative:

$$inv\ s \Longrightarrow 0 \le \Phi\ s$$

The potential of a data structure represents the savings that can pay for future restructurings of that 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.

The amortized complexity of an operation is defined as the actual cost plus the difference in potential (where *sum_list* adds up the elements of a list):

$$acost :: 'op \to 's\ list \to real$$
$$acost\ f\ ss = cost\ f\ ss + \Phi\ (exec\ f\ ss) - sum\_list\ (map\ \Phi\ ss)$$

The essential prerequisite for amortized analysis is that data structures are used in a single-threaded (or linear) manner [38], i.e. intermediate data structures cannot be used more than once. To capture this in a functional model we assume that the graph of operation calls is a tree. Formally, each node $T\ f\ ts$ consists of an operation symbol $f :: 'op$ and a list of subtrees ts.

Let us now analyze the complexity of executing such a tree of operations. We restrict to well-formed trees that respect the arity of each operation:

$$wf\ (T\ f\ ts) = (|ts| = arity\ f \wedge (\forall t \in set\ ts.\ wf\ t))$$

Function *state* returns the data structure resulting from the execution of a tree of operations:

$$state\ (T\ f\ ts) = exec\ f\ (map\ state\ ts)$$

We sum the actual and the amortized costs over a tree of operations:

$$acost\_sum\ (T\ f\ ts) = acost\ f\ (map\ state\ ts) + sum\_list\ (map\ acost\_sum\ ts)$$
$$cost\_sum\ (T\ f\ ts) = cost\ f\ (map\ state\ ts) + sum\_list\ (map\ cost\_sum\ ts)$$

Induction yields

$$wf\ ot \Longrightarrow cost\_sum\ ot = acost\_sum\ ot - \Phi\ (state\ ot)$$

Because all operations preserve the invariant, the invariant holds for *state ot* and thus $0 \leq \Phi\ (state\ ot)$ by assumption on Φ. Hence the amortized complexity is an upper bound of the real complexity:

$$wf\ ot \Longrightarrow cost\_sum\ ot \leq acost\_sum\ ot$$

To complete our formalization we add one more parameter, an explicit upper bound for the amortized complexity of each operation

$$U :: {}'op \Rightarrow {}'s\ list \Rightarrow real$$

subject to the following assumption:

$$(\forall s{\in}set\ ss.\ inv\ s) \wedge |ss| = arity\ f \Longrightarrow acost\ f\ ss \leq U\ f\ ss$$

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

$$wf\ ot \Longrightarrow cost\_sum\ ot \leq U\_sum\ ot \qquad (1)$$

where *U_sum* is defined like *cost_sum* and *acost_sum*.

So far we have pretended that the computation whose complexity we want to bound produces a single value and can therefore be represented by a tree. In general there may be multiple computations that do not all join up. This situation is captured by a multiset of trees of operations. The generalization of (1) to multisets is a trivial consequence, where $\in_\#$ is multiset membership:

$$\forall ot \in_\# M.\ wf\ ot \Longrightarrow (\sum ot \in_\# M.\ cost\_sum\ ot) \leq (\sum ot \in_\# M.\ U\_sum\ ot)$$

Note that our theory caters only for operations with result type $'s$. So-called *observers*, operations that return a different type, e.g. *bool*, are not suitable for amortized analysis: there is no sequence of operations that act on a succession of data structures because observers produce no new data structure. Hence observers require worst-case analysis. In all our examples their worst-case running time is constant. Sometimes an operation returns both a new data structure and an observable value. For example, the *delete minimum* operation on a priority queue may return both the new priority queue and the minimum element. This combination of a mutator and observer can be modeled by dropping the observer result (but not its computation) because the result is irrelevant for the complexity analysis. Now amortized analysis applies again.

Instantiating this theory of amortized complexity means defining the parameters and proving the assumptions above: all operations preserve *inv*, Φ is non-negative, and U is an upper bound of the amortized cost of each operation. Clearly the key property is the assumption about U. Our framework does not perform these proofs automatically but in yields the following benefits:

– It ensures that the user proves exactly the right properties.
– It yields theorem (1) that bounds the cost of a computation tree involving a certain data structure. This theorem may then be used in the complexity analysis of an application that uses this data structure.

Thus the framework is similar to a verification condition generator.

4 Easy Examples

The primary purpose of this section is to demonstrate the instantiation of the framework. The examples come from a standard textbook [7], except for the functional queue implementation (e.g. [38]). We do not discuss the proofs because they can be found in many textbooks. Isabelle's `auto` proof method (which combines rewriting with a bit of first-order reasoning) performs them automatically.

4.1 Binary Counter

We begin with the classic binary counter where incrementation can take linear time in the worst case but takes only constant amortized time because the worst case is rare. The state space $'s$ is just a list of booleans, starting with the least significant bit. There are two operations, initialization and increment, with arities 0 and 1:

$$\textbf{datatype}\ op = Empty \mid Incr$$

$$arity\ Empty = 0$$
$$arity\ Incr = 1$$

In the rest of the paper we do not show the *arity* function as it is always obvious.

Function *exec* maps syntax to semantics. In other words, it is a dispatcher that calls the actual functions associated with each operation symbol. The initialization function is inlined

$$exec\ Empty\ [] = []$$
$$exec\ Incr\ [bs] = incr\ bs$$

whereas *incr* is defined recursively:

$$incr\ [] = [True]$$
$$incr\ (False \cdot bs) = True \cdot bs$$
$$incr\ (True \cdot bs) = False \cdot incr\ bs$$

Analogously we have the *cost* function

$$cost\ Empty\ \_ = 1$$
$$cost\ Incr\ [bs] = t_{incr}\ bs$$

that dispatches to t_{incr} which is defined in complete analogy to *incr*:

$$t_{incr}\ [] = 1$$
$$t_{incr}\ (False \cdot \_) = 1$$
$$t_{incr}\ (True \cdot bs) = t_{incr}\ bs + 1$$

The (upper bounds for the) amortized complexities are constant:

$$U\ Empty\ \_ = 1$$
$$U\ Incr\ \_ = 2$$

The potential is the number of *True* bits:

$$\Phi\ bs = |filter\ id\ bs|$$

The higher the potential, the longer an increment may take (roughly speaking). This concludes the instantiation of the parameters of our amortized analysis theory. It remains to show that 2 is indeed an upper bound for the amortized complexity of *incr*. This follows immediately from

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

which is proved by induction. Isabelle needs to be told to perform induction on *bs* but the base case and induction step are proved automatically. It is the only theorem in this section that requires induction.

4.2 Dynamic Tables

Dynamic tables are tables where elements are added and deleted and the table grows and shrinks accordingly. At any point the table has a fixed size l and contains $n \le l$ elements. If the table overflows (upon insertion), the contents is copied into a larger table; if it underflows (upon deletion), the contents is copied into a smaller table. We ignore the actual elements because they are irrelevant for the complexity analysis. Therefore the operations

datatype *op* = *Empty* | *Ins* | *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. That means we define an abstract model rather than real code. This is how the operations behave:

$exec\ Empty\ [] = (0, 0)$
$exec\ Ins\ [(n, l)] = (n + 1, \textsf{if}\ n < l\ \textsf{then}\ l\ \textsf{else if}\ l = 0\ \textsf{then}\ 1\ \textsf{else}\ 2 * l)$
$exec\ Del\ [(n, l)] =$
$(n - 1, \textsf{if}\ n \le 1\ \textsf{then}\ 0\ \textsf{else if}\ 4 * (n - 1) < l\ \textsf{then}\ l\ div\ 2\ \textsf{else}\ l)$

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 *cost* function:

$cost\ Empty\ \_ = 1$
$cost\ Ins\ [(n, l)] = (\textsf{if}\ n < l\ \textsf{then}\ 1\ \textsf{else}\ n + 1)$
$cost\ Del\ [(n, l)] = (\textsf{if}\ n \le 1\ \textsf{then}\ 1\ \textsf{else if}\ 4 * (n - 1) < l\ \textsf{then}\ n\ \textsf{else}\ 1)$

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

We did not show the invariant for the binary counter because it is *True*. This time we have a non-trivial invariant:

$$inv\ (n,l) = (\textsf{if}\ l = 0\ \textsf{then}\ n = 0\ \textsf{else}\ n \le l \wedge l \le 4 * n)$$

The potential is also more complicated than before:

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

The amortized complexity bounds are:

$U\ Empty\ \_ = 1$
$U\ Ins\ \_ = 3$
$U\ Del\ \_ = 2$

4.3 Queues

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

datatype *'a op = Empty | Enq 'a | Deq*

A simple yet amortized constant time implementation of functional queues consists of two lists (stacks) (xs,ys):

```
$adjust\ (xs,\ ys) = (\textsf{if}\ ys = []\ \textsf{then}\ ([],\ rev\ xs)\ \textsf{else}\ (xs,\ ys))$

$exec\ Empty\ [] = ([],\ [])$
$exec\ (Enq\ x)\ [(xs,\ ys)] = adjust\ (x \cdot xs,\ ys)$
$exec\ Deq\ [(xs,\ ys)] = adjust\ (xs,\ tl\ ys)$

$cost\ Empty\ \_ = 0$
$cost\ (Enq\ \_)\ [(xs,\ ys)] = 1 + (\textsf{if}\ ys = []\ \textsf{then}\ |xs| + 1\ \textsf{else}\ 0)$
$cost\ Deq\ [(xs,\ ys)] = (\textsf{if}\ tl\ ys = []\ \textsf{then}\ |xs|\ \textsf{else}\ 0)$
```

Function *tl* takes the tail and *rev* reverses a list. The cost function counts only allocations of list cells and assumes *rev* has linear complexity.

The potential and the amortized complexities are

```
$\Phi\ (xs,ys) = |xs|$
$U\ Empty\ \_ = 0$
$U\ (Enq\ \_)\ \_ = 2$
$U\ Deq\ \_ = 0$
```

We ignore the observer operation that returns the oldest item, the head of *ys*, which is a worst-case constant time operation.

5 Skew Heaps

This section analyzes a beautifully simple data structure for priority queues: skew heaps [42]. Heaps are trees where the least element in each subtree is at the root. Skew heaps provide the following operations:

datatype *'a op = Empty | Insert 'a | Del_min | Merge*

We ignore the observer operation that returns the minimal element because it runs in worst-case constant time and does not modify the heap.

The central operation on skew heaps is *merge*. It merges two skew heaps and swaps children along the merge path:

```
$merge\ h_1\ h_2 =$
$(\textsf{case}\ h_1\ \textsf{of}\ \langle\rangle \Rightarrow h_2$
 $|\ \langle l_1,\ a_1,\ r_1\rangle \Rightarrow$
    $\textsf{case}\ h_2\ \textsf{of}\ \langle\rangle \Rightarrow h_1$
    $|\ \langle l_2,\ a_2,\ r_2\rangle \Rightarrow$
       $\textsf{if}\ a_1 \le a_2\ \textsf{then}\ \langle merge\ h_2\ r_1,\ a_1,\ l_1\rangle\ \textsf{else}\ \langle merge\ h_1\ r_2,\ a_2,\ l_2\rangle)$
```

The remaining operations are implemented via *merge*:

$$\begin{aligned}
&\textit{exec Empty}\ [] = \langle\rangle\\
&\textit{exec}\ (\textit{Insert a})\ [h] = \textit{merge}\ \langle\langle\rangle, a, \langle\rangle\rangle\ h\\
&\textit{exec Del\_min}\ [h] = \textit{del\_min h}\\
&\textit{exec Merge}\ [h_1, h_2] = \textit{merge}\ h_1\ h_2
\end{aligned}$$

$$\begin{aligned}
&\textit{del\_min}\ \langle\rangle = \langle\rangle\\
&\textit{del\_min}\ \langle l, m, r\rangle = \textit{merge l r}
\end{aligned}$$

For the functional correctness proofs see [30].

Sleator and Tarjan show that merge (and hence insert and *del_min*) has amortized complexity of at most $3\log_2 n$. Their proof is mostly prose. Our amortized analysis is based on the precise functional account by Kaldewaij and Schoenmakers [23] but with three differences:

- Kaldewaij and Schoenmakers reduce the multiplicative constant from 3 to 1.44. This complicates their proof considerably. We simplify their proof by aiming only for the original factor of 3.
- Kaldewaij and Schoenmakers phrase their proof as a synthesis of Φ, we simplify the proceedings by verifying a given fixed Φ.
- Our *merge* differs from theirs in that it stops as soon as one of the two arguments is empty.

We adopt their cost measure that counts (in essence) the number of calls of *merge*:

$$\begin{aligned}
&t_{merge}\ \langle\rangle\ h = 1\\
&t_{merge}\ h\ \langle\rangle = 1\\
&t_{merge}\ \langle l_1, a_1, r_1\rangle\ \langle l_2, a_2, r_2\rangle =\\
&(\text{if } a_1 \le a_2 \text{ then } t_{merge}\ \langle l_2, a_2, r_2\rangle\ r_1 \text{ else } t_{merge}\ \langle l_1, a_1, r_1\rangle\ r_2) + 1
\end{aligned}$$

$$\begin{aligned}
&\textit{cost Empty}\ [] = 1\\
&\textit{cost}\ (\textit{Insert a})\ [h] = t_{merge}\ \langle\langle\rangle, a, \langle\rangle\rangle\ h\\
&\textit{cost Del\_min}\ [h] = (\text{case } h \text{ of } \langle\rangle \Rightarrow 1\\
&\qquad\qquad\qquad\qquad \mid \langle t_1, a, t_2\rangle \Rightarrow t_{merge}\ t_1\ t_2)\\
&\textit{cost Merge}\ [h_1, h_2] = t_{merge}\ h_1\ h_2
\end{aligned}$$

Our potential function is an instance of the one by Kaldewaij and Schoenmakers: it counts the number of “right heavy” nodes. The reason is that *merge* descends along the right spine and therefore right heavy nodes are bad.

$$\textit{rh l r} = (\text{if } |l| < |r| \text{ then } 1 \text{ else } 0)$$

$$\Phi\ \langle\rangle = 0 \qquad \Phi\ \langle l, \_, r\rangle = \Phi\ l + \Phi\ r + \textit{rh l r}$$

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

$$\begin{aligned}
&\Gamma\ \langle\rangle = 0 \qquad \Gamma\ \langle l, \_, r\rangle = \textit{rh l r} + \Gamma\ l\\
&\Delta\ \langle\rangle = 0 \qquad \Delta\ \langle l, \_, r\rangle = 1 - \textit{rh l r} + \Delta\ r
\end{aligned}$$

We will need the following two properties below:

$$\Gamma\ h \le \log_2 |h|_1 \tag{2}$$
$$\Delta\ h \le \log_2 |h|_1 \tag{3}$$

On paper one can show them easily by induction. However, Isabelle is not able to prove the induction step automatically. Hence we apply a simple and frequently successful recipe: remove logarithms by exponentiation. That is, we prove $2^{\Gamma\ h} \le |h| + 1$ and $2^{\Delta\ h} \le |h| + 1$ by induction and now the base case and induction step are proved automatically, as are the consequences (2) and (3).

A third property is also be proved automatically by Isabelle but is less intuitive. Hence we show the details.

Lemma 5.1
$t_{merge}\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2 \le \Gamma\ (merge\ t_1\ t_2) + \Delta\ t_1 + \Delta\ t_2 + 1$

Proof by induction on the computation of *merge*. We consider only the node-node case: let $t_1 = \langle l_1, a_1, r_1\rangle$ and $t_2 = \langle l_2, a_2, r_2\rangle$. W.l.o.g. assume $a_1 \le a_2$. Let $m = merge\ t_2\ r_1$.

$$\begin{aligned}
&t_{merge}\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2\\
&= t_{merge}\ t_2\ r_1 + 1 + \Phi\ m + \Phi\ l_1 + rh\ m\ l_1 - \Phi\ t_1 - \Phi\ t_2\\
&= t_{merge}\ t_2\ r_1 + 1 + \Phi\ m + rh\ m\ l_1 - \Phi\ r_1 - rh\ l_1\ r_1 - \Phi\ t_2\\
&\le \Gamma\ m + \Delta\ t_2 + \Delta\ r_1 + rh\ m\ l_1 + 2 - rh\ l_1\ r_1 && \text{by IH}\\
&= \Gamma\ m + \Delta\ t_2 + \Delta\ t_1 + rh\ m\ l_1 + 1\\
&= \Gamma\ (merge\ t_1\ t_2) + \Delta\ t_1 + \Delta\ t_2 + 1
\end{aligned}$$

Now the logarithmic amortized complexity of *merge* follows:

$$\begin{aligned}
&t_{merge}\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2\\
&\le \Gamma\ (merge\ t_1\ t_2) + \Delta\ t_1 + \Delta\ t_2 + 1 && \text{by Lemma 5.1}\\
&\le \log_2 |merge\ t_1\ t_2|_1 + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1 && \text{by (2), (3)}\\
&= \log_2 (|t_1|_1 + |t_2|_1 - 1) + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1\\
& && \text{because } |merge\ t_1\ t_2| = |t_1| + |t_2|\\
&\le \log_2 (|t_1|_1 + |t_2|_1) + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1\\
&\le \log_2 (|t_1|_1 + |t_2|_1) + 2 * \log_2 (|t_1|_1 + |t_2|_1) + 1\\
& && \text{because } \log_b x + \log_b y \le 2 * \log_b (x + y) \text{ if } x,y > 0\\
&= 3 * \log_2 (|t_1|_1 + |t_2|_1) + 1
\end{aligned}$$

Now it is easy to verify the following upper bounds:

$$\begin{aligned}
&U\ Empty\ [] = 1\\
&U\ (Insert\ \_)\ [h] = 3 * \log_2 (|h|_1 + 2) + 1\\
&U\ Del\_min\ [h] = 3 * \log_2 (|h|_1 + 2) + 3\\
&U\ Merge\ [h_1, h_2] = 3 * \log_2 (|h_1|_1 + |h_2|_1) + 1
\end{aligned}$$

Note that Isabelle supports implicit coercions, in particular from *nat* to *real*, that are inserted automatically [44].

In the calculation above we have used that $\log_2 x + \log_2 y \le 2 * \log_2 (x + y)$ (if $x,y > 0$). It, and a number of similar lemmas $l \le r$ used below, can all be proved by showing $2^l \le 2^r$: $2^{\log_2 x + \log_2 y} = x * y \le (x + y)^2 = 2^{2 * \log_2 (x + y)}$. This is another instance of the above recipe to remove logarithms by exponentiation.

```
$splay\ x\ \langle\rangle = \langle\rangle$
$splay\ x\ \langle AB, b, CD\rangle =$
(if $x = b$ then $\langle AB, b, CD\rangle$
 else if $x < b$
      then case $AB$ of  $\langle\rangle \Rightarrow \langle AB, b, CD\rangle$
            | $\langle A, a, B\rangle \Rightarrow$
               if $x = a$ then $\langle A, a, \langle B, b, CD\rangle\rangle$
               else if $x < a$
                    then if $A = \langle\rangle$ then $\langle A, a, \langle B, b, CD\rangle\rangle$
                         else case $splay\ x\ A$ of  $\langle A_1, a', A_2\rangle \Rightarrow \langle A_1, a', \langle A_2, a, \langle B, b, CD\rangle\rangle\rangle$
                    else if $B = \langle\rangle$ then $\langle A, a, \langle B, b, CD\rangle\rangle$
                         else case $splay\ x\ B$ of  $\langle B_1, b', B_2\rangle \Rightarrow \langle\langle A, a, B_1\rangle, b', \langle B_2, b, CD\rangle\rangle$
      else case $CD$ of  $\langle\rangle \Rightarrow \langle AB, b, CD\rangle$
           | $\langle C, c, D\rangle \Rightarrow$
              if $x = c$ then $\langle\langle AB, b, C\rangle, c, D\rangle$
              else if $x < c$
                   then if $C = \langle\rangle$ then $\langle\langle AB, b, C\rangle, c, D\rangle$
                        else case $splay\ x\ C$ of  $\langle C_1, c', C_2\rangle \Rightarrow \langle\langle AB, b, C_1\rangle, c', \langle C_2, c, D\rangle\rangle$
                   else if $D = \langle\rangle$ then $\langle\langle AB, b, C\rangle, c, D\rangle$
                        else case $splay\ x\ D$ of  $\langle D_1, d, D_2\rangle \Rightarrow \langle\langle\langle AB, b, C\rangle, c, D_1\rangle, d, D_2\rangle)$
```

Fig. 1 Function *splay*

6 Splay Trees

A splay tree [41] 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 $'a \to 'a\ tree \to 'a\ tree$. It searches some element x of a linearly ordered type $'a$; on the way back up it rotates x to the root of the tree. Most presentations of *splay* confine themselves to the 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 Figure 1.

Although it is perfectly possible to work with the definition in Figure 1, Isabelle's function definition command [24] proves an induction theorem that has only two cases, leaf and node, corresponding to the two equations for *splay*. If a proof about *splay* needs to examine further subcases, this needs to be done explicitly in the proof text (unless the proof is automatic). In the worst case this leads to as many manual case analyses as there are conditionals on the right-hand side of the equations. Luckily the function definition command offers a better alternative: define the function by conditional equations and obtain an induction principle with a separate case for each defining equation. This is our actual definition and is shown in Figure 2, the one in Figure 1 is merely a direct consequence. The resulting induction theorem leads to more compact proofs (one third shorter in one instance) by reducing boiler plate proof steps. Schoenmakers [40] also defines *splay* by conditional equations.

Given splaying, searching for an element in the tree is trivial: splay with the given element and check if it ends up at the root. For insertion and deletion, algorithm texts often show pictures only. In contrast, we show the code only, in Figure 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). To delete a, you splay with a and if a ends up at the root, you replace it with the maximal element removed

$$\begin{aligned}
& \mathit{splay}\ x\ \langle\rangle = \langle\rangle \\
& \mathit{splay}\ x\ \langle A, x, B\rangle = \langle A, x, B\rangle \\
x < b \Longrightarrow{} & \mathit{splay}\ x\ \langle\langle A, x, B\rangle, b, C\rangle = \langle A, x, \langle B, b, C\rangle\rangle \\
x < b \Longrightarrow{} & \mathit{splay}\ x\ \langle\langle\rangle, b, A\rangle = \langle\langle\rangle, b, A\rangle \\
x < a \wedge x < b \Longrightarrow{} & \mathit{splay}\ x\ \langle\langle\langle\rangle, a, A\rangle, b, B\rangle = \langle\langle\rangle, a, \langle A, b, B\rangle\rangle \\
x < b \wedge x < c \wedge AB \neq \langle\rangle \Longrightarrow{} & \mathit{splay}\ x\ \langle\langle AB, b, C\rangle, c, D\rangle \\
& = \mathsf{case}\ \mathit{splay}\ x\ AB\ \mathsf{of}\ \langle A, a, B\rangle \Rightarrow \langle A, a, \langle B, b, \langle C, c, D\rangle\rangle\rangle \\
a < x \wedge x < b \Longrightarrow{} & \mathit{splay}\ x\ \langle\langle A, a, \langle\rangle\rangle, b, B\rangle = \langle A, a, \langle\langle\rangle, b, B\rangle\rangle \\
a < x \wedge x < c \wedge BC \neq \langle\rangle \Longrightarrow{} & \mathit{splay}\ x\ \langle\langle A, a, BC\rangle, c, D\rangle \\
& = \mathsf{case}\ \mathit{splay}\ x\ BC\ \mathsf{of}\ \langle B, b, C\rangle \Rightarrow \langle\langle A, a, B\rangle, b, \langle C, c, D\rangle\rangle \\
a < x \Longrightarrow{} & \mathit{splay}\ x\ \langle A, a, \langle B, x, C\rangle\rangle = \langle\langle A, a, B\rangle, x, C\rangle \\
a < x \Longrightarrow{} & \mathit{splay}\ x\ \langle A, a, \langle\rangle\rangle = \langle A, a, \langle\rangle\rangle \\
a < x \wedge x < c \wedge BC \neq \langle\rangle \Longrightarrow{} & \mathit{splay}\ x\ \langle A, a, \langle BC, c, D\rangle\rangle \\
& = \mathsf{case}\ \mathit{splay}\ x\ BC\ \mathsf{of}\ \langle B, b, C\rangle \Rightarrow \langle\langle A, a, B\rangle, b, \langle C, c, D\rangle\rangle \\
a < x \wedge x < b \Longrightarrow{} & \mathit{splay}\ x\ \langle A, a, \langle\langle\rangle, b, C\rangle\rangle = \langle\langle A, a, \langle\rangle\rangle, b, C\rangle \\
a < x \wedge b < x \Longrightarrow{} & \mathit{splay}\ x\ \langle A, a, \langle B, b, \langle\rangle\rangle\rangle = \langle\langle A, a, B\rangle, b, \langle\rangle\rangle \\
a < x \wedge b < x \wedge CD \neq \langle\rangle \Longrightarrow{} & \mathit{splay}\ x\ \langle A, a, \langle B, b, CD\rangle\rangle \\
& = \mathsf{case}\ \mathit{splay}\ x\ CD\ \mathsf{of}\ \langle C, c, D\rangle \Rightarrow \langle\langle\langle A, a, B\rangle, b, C\rangle, c, D\rangle
\end{aligned}$$

Fig. 2 Conditional definition of function *splay*

```
$insert\ x\ t =$
(if $t = \langle\rangle$ then $\langle\langle\rangle, x, \langle\rangle\rangle$
 else case $splay\ x\ t$ of
     $\langle l, a, r\rangle \Rightarrow$ if $x < a$ then $\langle l, x, \langle\langle\rangle, a, r\rangle\rangle$ else if $a < x$ then $\langle\langle l, a, \langle\rangle\rangle, x, r\rangle$ else $\langle l, a, r\rangle$)

$delete\ x\ t =$
(if $t = \langle\rangle$ then $\langle\rangle$
 else case $splay\ x\ t$ of
     $\langle l, a, r\rangle \Rightarrow$
       if $x = a$ then if $l = \langle\rangle$ then $r$ else case $splay\_max\ l$ of $\langle l', m, r'\rangle \Rightarrow \langle l', m, r\rangle$
       else $\langle l, a, r\rangle$)

$splay\_max\ \langle\rangle = \langle\rangle$
$splay\_max\ \langle A, a, \langle\rangle\rangle = \langle A, a, \langle\rangle\rangle$
$splay\_max\ \langle A, a, \langle B, b, CD\rangle\rangle =$
(if $CD = \langle\rangle$ then $\langle\langle A, a, B\rangle, b, \langle\rangle\rangle$
 else case $splay\_max\ CD$ of $\langle C, c, D\rangle \Rightarrow \langle\langle\langle A, a, B\rangle, b, C\rangle, c, D\rangle$)
```

Fig. 3 Functions *insert*, *delete* and *splay_max*

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 have ignored functional correctness but for splay trees we actually need elements of it in the verification of the complexity. 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:

$$\mathit{set\_tree}\ (\mathit{splay}\ a\ t) = \mathit{set\_tree}\ t \qquad \mathit{bst}\ t \Longrightarrow \mathit{bst}\ (\mathit{splay}\ a\ t)$$

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

$$bst\ t \Longrightarrow set\_tree\ (delete\ a\ t) = set\_tree\ t - \{a\}$$

Automatic proofs of functional correctness are presented elsewhere [33]. In the complexity proofs below we do not actually use functional correctness but we use *bst* and *set_tree* and the fact that *bst* is an invariant.

Now we present two amortized analyses: a simpler one that yields the bounds proved by Sleator and Tarjan [41] and a more complicated and precise one due to Schoenmakers [40].

6.2 Amortized Analysis

The timing functions shown in Figure 4 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, a_{splay}, is defined as usual:

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

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

$$a_{splay}\ a\ t \le 3 * (\varphi\ t - \varphi\ \langle l, a, r\rangle) + 1 \tag{4}$$

Let us look at one case of the inductive proof in detail. We pick the so-called zig-zig case shown in Figure 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 $splay\ a\ R = \langle R_1, a, R_2\rangle =: R'$.

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

From (4) we obtain in the worst case ($l = r = \langle\rangle$):

$$bst\ t \wedge a \in set\_tree\ t \Longrightarrow a_{splay}\ a\ t \le 3 * (\varphi\ t - 1) + 1$$

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

$t_{splay\_max}\ \langle\rangle = 1$
$t_{splay\_max}\ \langle l, b, \langle\rangle\rangle = 1$
$t_{splay\_max}\ \langle l, b, \langle rl, c, rr\rangle\rangle =$ (if $rr = \langle\rangle$ then $1$ else $t_{splay\_max}\ rr + 1$)

$t_{delete}\ a\ t =$
(if $t = \langle\rangle$ then $0$
 else $t_{splay}\ a\ t\ +$
      (case $splay\ a\ t$ of
       $\langle l, x, r\rangle \Rightarrow$ if $x = a$ then case $l$ of $\langle\rangle \Rightarrow 0$
                                      | $\langle tree1, a, tree2\rangle \Rightarrow t_{splay\_max}\ l$
                     else $0$))
```

Fig. 4 Running time functions for splay trees

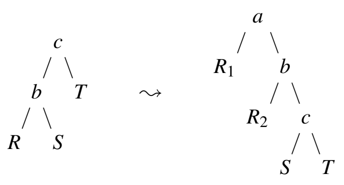

Fig. 5 Zig-zig case for *splay*: $a < b < c$

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:

$$bst\ t \Longrightarrow a_{splay}\ a\ t \leq 3 * \varphi\ t + 1 \tag{5}$$

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 a_{splay_max} of *splay_max*

$$a_{splay\_max}\ t = t_{splay\_max}\ t + \Phi\ (splay\_max\ t) - \Phi\ t$$

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

Now we can apply our amortized analysis theory:

datatype *'a op = Empty | Splay 'a | Insert 'a | Delete 'a*

$exec\ Empty\ [] = \langle\rangle$	$cost\ Empty\ [] = 1$
$exec\ (Splay\ a)\ [t] = splay\ a\ t$	$cost\ (Splay\ a)\ [t] = t_{splay}\ a\ t$
$exec\ (Insert\ a)\ [t] = insert\ a\ t$	$cost\ (Insert\ a)\ [t] = t_{splay}\ a\ t$
$exec\ (Delete\ a)\ [t] = delete\ a\ t$	$cost\ (Delete\ a)\ [t] = t_{delete}\ a\ t$

$U\ Empty\ [] = 1$
$U\ (Splay\ \_)\ [t] = 3 * \varphi\ t + 1$
$U\ (Insert\ \_)\ [t] = 4 * \varphi\ t + 2$
$U\ (Delete\ \_)\ [t] = 6 * \varphi\ t + 2$

The fact that the given U is indeed a correct upper bound follows from the upper bounds for a_{splay} and a_{splay_max}; for *Insert* and *Delete* the proof needs some more case distinctions and log-manipulations.

Note that we have not provided a function for searching an element in a tree because that function merely calls *splay* and checks if the given element has ended up at the root. Hence its complexity is the same as that of *splay*.

6.3 Improved Amortized Analysis

This subsection follows Schoenmakers [40] (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 constraints in [40] look different but are equivalent.

The definition of Φ now depends on α and β:

$$\varphi\ l\ r = \beta * \log_{\alpha}\ (|l|_1 + |r|_1)$$

$$\Phi\ \langle\rangle = 0$$
$$\Phi\ \langle l, \_, r\rangle = \Phi\ l + \Phi\ r + \varphi\ l\ r$$

Functions a_{splay} and a_{splay_max} are defined as before, but with the new Φ. The following upper bound is again proved by induction but this time with the help of the above constraints:

$$bst\ t \wedge \langle l, a, r\rangle \in subtrees\ t \Longrightarrow a_{splay}\ a\ t \leq \log_{\alpha}\ (|t|_1\ /\ (|l|_1 + |r|_1)) + 1$$

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

$bst\ t \Longrightarrow a_{splay}\ a\ t \leq \log_{\alpha}\ |t|_1 + 1$

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 (5):

$bst\ t \Longrightarrow a_{splay34}\ a\ t \leq 3\ /\ 2 * \varphi\ t + 1$

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 a_{splay_max34} as for $a_{splay34}$. 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:

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

The proofs in this subsection require highly nonlinear arithmetic. Only some of the polynomial inequalities can be automated with Harrison's sum-of-squares method [16].

7 Splay Heaps

Splay heaps are another self-adjusting data structure and were invented by Okasaki [38]. Splay heaps are organized internally like splay trees but they implement a priority queue interface without *Merge*. 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:

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

$partition\ p\ \langle\rangle = (\langle\rangle, \langle\rangle)$
$partition\ p\ \langle AB, ab, BC\rangle =$
$(\text{if}\ ab \leq p$
 $\text{then case}\ BC\ \text{of}\ \ \langle\rangle \Rightarrow (\langle AB, ab, BC\rangle, \langle\rangle)$
      $|\ \langle B, b, C\rangle \Rightarrow$
        $\text{if}\ b \leq p\ \text{then let}\ (C_1, C_2) = partition\ p\ C\ \text{in}\ (\langle\langle AB, ab, B\rangle, b, C_1\rangle, C_2)$
        $\text{else let}\ (B_1, B_2) = partition\ p\ B\ \text{in}\ (\langle AB, ab, B_1\rangle, \langle B_2, b, C\rangle)$
 $\text{else case}\ AB\ \text{of}\ \ \langle\rangle \Rightarrow (\langle\rangle, \langle AB, ab, BC\rangle)$
      $|\ \langle A, a, B\rangle \Rightarrow$
       $\text{if}\ a \leq p\ \text{then let}\ (B_1, B_2) = partition\ p\ B\ \text{in}\ (\langle A, a, B_1\rangle, \langle B_2, ab, BC\rangle)$
       $\text{else let}\ (A_1, A_2) = partition\ p\ A\ \text{in}\ (A_1, \langle A_2, a, \langle B, ab, BC\rangle\rangle))$
```

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

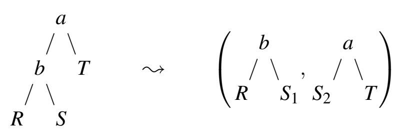

Fig. 6 Zig-zag case for *partition*: $b \le p < a$

$del\_min\ \langle\rangle = \langle\rangle$
$del\_min\ \langle\langle\rangle, \_, r\rangle = r$
$del\_min\ \langle\langle A, a, B\rangle, b, C\rangle =$
$(\mathsf{if}\ A = \langle\rangle\ \mathsf{then}\ \langle B, b, C\rangle\ \mathsf{else}\ \langle del\_min\ A, a, \langle B, b, C\rangle\rangle)$

In contrast to search trees, priority queues may contain elements multiple times. Therefore splay heaps satisfy *bst_wrt* $(\le)$ instead of *bst_wrt* $(<)$. It is an invariant for both *partition* and *del_min*:

$partition\ p\ t = (l, r) \wedge bst\_wrt\ (\le)\ t \Longrightarrow bst\_wrt\ (\le)\ \langle l, p, r\rangle$
$bst\_wrt\ (\le)\ t \Longrightarrow bst\_wrt\ (\le)\ (del\_min\ t)$

Under this invariant we can prove the correctness of *partition* and *del_min*:

$bst\_wrt\ (\le)\ t \wedge partition\ p\ t = (l, r) \Longrightarrow mset\_tree\ t = mset\_tree\ l + mset\_tree\ r$
$mset\_tree\ (del\_min\ h) = mset\_tree\ h - \{\#get\_min\ h\#\}$

where *mset_tree t* is the multiset of elements in tree t, $+$ and $-$ are union and difference of multisets, $\{\#x\#\}$ is a singleton multiset, and *get_min t* is the leftmost element of tree t. For the proofs see [31].

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 Section 6.2. We abbreviate the amortized complexity of a computation $partition\ p\ t = (l', r')$ by

$$\mathscr{A}\ p\ t = t_{part}\ p\ t + \Phi\ l' + \Phi\ r' - \Phi\ t$$

The following logarithmic bound on $\mathscr{A}\ p\ t$ is proved by computation induction on *partition p t*:

$$bst\_wrt\ (\le)\ t \Longrightarrow \mathscr{A}\ p\ t \le 2 * \varphi\ t + 1$$

Okasaki [38] shows the zig-zig case of the induction, we show the zig-zag case in Figure 6. Subtrees with root x are called X on the left and X' on the right-hand side. Thus Figure 6 depicts $partition\ p\ A = (B', A')$ assuming the recursive call $partition\ p\ S = (S_1, S_2)$.

$$\begin{aligned}
&\mathscr{A}\ p\ A = t_{part}\ p\ A + \Phi\ B' + \Phi\ A' - \Phi\ A && \text{by def. of } \mathscr{A}\ p\ A\\
&= 1 + t_{part}\ p\ S + \Phi\ B' + \Phi\ A' - \Phi\ A && \text{by def. of } t_{part}\\
&= 1 + \mathscr{A}\ p\ S - \Phi\ S_1 - \Phi\ S_2 + \Phi\ S + \Phi\ B' + \Phi\ A' - \Phi\ A && \text{by def. of } \mathscr{A}\ p\ S
\end{aligned}$$

$= 1 + \mathscr{A}\ p\ S + \varphi\ B' + \varphi\ A' - \varphi\ B - \varphi\ A$ by def. of Φ
$\leq 2 * \varphi\ S + 2 + \varphi\ B' + \varphi\ A' - \varphi\ B - \varphi\ A$ by IH
$< 2 + \varphi\ B' + \varphi\ A'$ because $\varphi\ S < \varphi\ B$ and $\varphi\ S < \varphi\ A$
$\leq 2 * \log_2\ (|R|_1 + |S_1|_1 + |S_2|_1 + |T|_1 - 1) + 1$
because $1 + \log_2 x + \log_2 y \leq 2 * \log_2\ (x + y - 1)$ if $x,y \geq 2$
$= 2 * \varphi\ A + 1$ because $|S_1| + |S_2| = |S|$

The proof of the amortized complexity of *del_min* is similar to the proof for *splay_max*: $t_{dm}\ t + \Phi\ (del\_min\ t) - \Phi\ t \leq 2 * \varphi\ t + 1$. Now it is routine to verify the following amortized complexities by instantiating our standard theory:

exec Empty $[] = \langle\rangle$ *cost Empty* $[] = 1$
exec $(Insert\ a)\ [t] = insert\ a\ t$ *cost* $(Insert\ a)\ [t] = t_{part}\ a\ t$
exec Del_min $[t] = del\_min\ t$ *cost Del_min* $[t] = t_{dm}\ t$

U Empty $[] = 1$
$U\ (Insert\ \_)\ [t] = 3 * \log_2\ (|t|_1 + 1) + 1$
$U\ Del\_min\ [t] = 2 * \varphi\ t + 1$

8 Pairing Heaps

This section analyzes another easy-to-implement data structure for priority queues: pairing heaps [13]. We follow the data structures, algorithms and roughly also the proofs of the original publication, except that our implementation is not imperative but functional.

Pairing heaps are represented as binary trees subject to a simple invariant: the root is either empty or a node whose right child is empty:

$is\_root\ h = (\mathsf{case}\ h\ \mathsf{of}\ \langle\rangle \Rightarrow True \mid \langle l, x, r\rangle \Rightarrow r = \langle\rangle)$

The left child can be viewed as a list (in binary tree form) of pairing heaps. Moreover, a pairing heap should satisfy the heap condition:

$pheap\ \langle\rangle = True$
$pheap\ \langle l, x, r\rangle = (pheap\ l \wedge pheap\ r \wedge (\forall y{\in}set\_tree\ l.\ x \leq y))$

It turns out that the proof of functional correctness requires both invariants, the proof of amortized complexity only needs *is_root*.

The central operation on pairing heaps is *link*. Called with the root of a pairing heap it behaves like the identity function; called with a heap that has a non-empty right child it permutes some nodes and subtrees (see Figure 7). Viewing *link* as an operation on a list (in binary tree form) of heaps it merges the first two heaps in the list by moving the one with the bigger root into the list of the other one.

$link\ \langle\rangle = \langle\rangle$
$link\ \langle lx, x, \langle\rangle\rangle = \langle lx, x, \langle\rangle\rangle$
$link\ \langle lx, x, \langle ly, y, ry\rangle\rangle =$
$(\mathsf{if}\ x < y\ \mathsf{then}\ \langle\langle ly, y, lx\rangle, x, ry\rangle\ \mathsf{else}\ \langle\langle lx, x, ly\rangle, y, ry\rangle)$

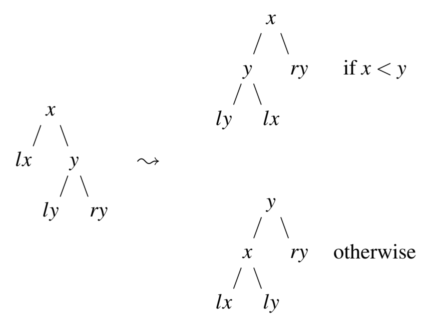

Fig. 7 *link* visualized

Inserting an element is realized like for skew heaps

$insert\ x\ h = merge\ \langle\langle\rangle, x, \langle\rangle\rangle\ h$

where *merge* combines two heaps. Applied to two non-empty heaps, function *merge* works in two steps: make one heap the right child of the root of the other heap, then call *link*.

$merge\ \langle\rangle\ h = h$
$merge\ h\ \langle\rangle = h$
$merge\ \langle lx, x, \langle\rangle\rangle\ \langle ly, y, \langle\rangle\rangle = link\ \langle lx, x, \langle ly, y, \langle\rangle\rangle\rangle$

Let us now focus on *del_min* which is defined as follows:

$del\_min\ \langle\rangle = \langle\rangle$
$del\_min\ \langle l, \_, \langle\rangle\rangle = pass_2\ (pass_1\ l)$

$pass_1\ \langle\rangle = \langle\rangle$
$pass_1\ \langle lx, x, \langle\rangle\rangle = \langle lx, x, \langle\rangle\rangle$
$pass_1\ \langle lx, x, \langle ly, y, ry\rangle\rangle = link\ \langle lx, x, \langle ly, y, pass_1\ ry\rangle\rangle$

$pass_2\ \langle\rangle = \langle\rangle$
$pass_2\ \langle l, x, r\rangle = link\ \langle l, x, pass_2\ r\rangle$

Functions $pass_1/pass_2$ call *link* on every/every second node of the sequence of nodes obtained by descending to the right from the root. Both operations work in a bottom-up manner (otherwise the complexity is worse [13]). Note that the only purpose of $pass_1$ is the complexity: without $pass_1$, *del_min* is still correct but the amortized running times would be worse. Also note that both passes can be performed in a single pass. We call this function *merge_pairs*:

$merge\_pairs\ \langle\rangle = \langle\rangle$
$merge\_pairs\ \langle lx, x, \langle\rangle\rangle = \langle lx, x, \langle\rangle\rangle$
$merge\_pairs\ \langle lx, x, \langle ly, y, ry\rangle\rangle = link\ (link\ \langle lx, x, \langle ly, y, merge\_pairs\ ry\rangle\rangle)$

It can easily be shown that

$pass_2\ (pass_1\ hs) = merge\_pairs\ hs$

Function *merge_pairs* is more efficient than $pass_2 \circ pass_1$ by a constant factor, but we follow [13] in separating the two passes because this seems to simplify the amortized analysis.

The proof of functional correctness of pairing heaps is straightforward and can be found elsewhere [3].

8.1 Amortized Analysis

The potential function is defined as

$$\Phi\ \langle\rangle = 0$$
$$\Phi\ \langle l,\ \_,\ r\rangle = \Phi\ l + \Phi\ r + \log_2\ (1 + |l| + |r|)$$

Function *insert* can cause an at most logarithmic change in potential:

$$\mathit{is\_root}\ h \Longrightarrow \Phi\ (\mathit{insert}\ x\ h) - \Phi\ h \le \log_2\ (|h| + 1) \tag{6}$$

Because none of the functions are recursive, this proof is automatic.

Now we analyze *del_min*. Its running time is linear in the number of nodes reachable by descending to the right (starting from the left child of the root). We denote this metric by $|\_|_R$:

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

Therefore we have to show that the potential change compensates for this linear work. Our main goal is now to show that if $lx \neq \langle\rangle$ then

$$\Phi\ (\mathit{del\_min}\ \langle lx,\ x,\ \langle\rangle\rangle) - \Phi\ \langle lx,\ x,\ \langle\rangle\rangle \le 3 * \log_2\ |lx| - |lx|_R + 2$$

This will be done roughly in two steps: First we show that $pass_1$ frees enough potential to compensate for the work linear in $|lx|_R$ and increases the potential only by a logarithmic term. Then we show that the increase due to $pass_2$ is also only at most logarithmic. Combining these results one easily shows that the amortized running time of *del_min* is indeed logarithmic.

In the proofs below note that

$$|pass_1\ h| = |h| \qquad |pass_2\ h| = |h|$$

We define a recursive upper bound *upperbound* for the potential change of $pass_1$

$$\mathit{upperbound}\ \langle\rangle = 0$$
$$\mathit{upperbound}\ \langle\_,\ \_,\ \langle\rangle\rangle = 0$$
$$\mathit{upperbound}\ \langle lx,\ \_,\ \langle ly,\ \_,\ \langle\rangle\rangle\rangle = 2 * \log_2\ (|lx| + |ly| + 2)$$
$$\mathit{upperbound}\ \langle lx,\ \_,\ \langle ly,\ \_,\ ry\rangle\rangle =$$
$$2 * \log_2\ (|lx| + |ly| + |ry| + 2) - 2 * \log_2\ |ry| - 2 + \mathit{upperbound}\ ry$$

and show that it is indeed an upper bound:

Lemma 8.1 $\Phi\ (pass_1\ hs) - \Phi\ hs \le \mathit{upperbound}\ hs$

Proof by induction on the computation of *upperbound*. Thus there is one case for each defining equation. Cases 1 and 2 are trivial. Case 3 needs some simple arithmetic manipulations. Given $hs = \langle lx,\ x,\ \langle ly,\ y,\ \langle\rangle\rangle\rangle$ we can deduce

$$\begin{aligned}&\Phi\ (pass_1\ hs) - \Phi\ hs = \log_2\ (|lx| + |ly| + 1) - \log_2\ (|ly| + 1)\\ &\le 2 * \log_2\ (|lx| + |ly| + 2) = upperbound\ hs\end{aligned}$$

For case 4, the induction step, we have $hs = \langle lx, x, rx\rangle$, $rx = \langle ly, y, ry\rangle$ and $ry \neq \langle\rangle$. First note

$$\begin{aligned}&\log_2\ (|lx| + |ly| + 1) + 2 * \log_2\ |ry| + 2\\ &\le 2 * \log_2\ (|lx| + |ly| + |ry| + 1) + \log_2\ |ry|\\ &\qquad\text{because } \log_2 x + \log_2 y + 2 \le 2 * \log_2\ (x + y) \text{ if } x,y > 0\\ &\le 2 * \log_2\ (|lx| + |ly| + |ry| + 2) + \log_2\ |ry| \qquad \text{by monotonicity of log}\\ &= 2 * \log_2\ |hs| + \log_2\ |ry|\\ &\le 2 * \log_2\ |hs| + \log_2\ (|ly| + |ry| + 1) \qquad \text{by monotonicity of log}\end{aligned}$$

which implies

$$\begin{aligned}&\log_2\ (|lx| + |ly| + 1) - \log_2\ (|ly| + |ry| + 1)\\ &\le 2 * \log_2\ |hs| - 2 * \log_2\ |ry| - 2\\ &= upperbound\ hs - upperbound\ ry \qquad (*)\end{aligned}$$

Using the definitions we obtain

$$\begin{aligned}&\Phi\ (pass_1\ hs) - \Phi\ hs\\ &= \log_2\ (|lx| + |ly| + 1) - \log_2\ (|lx| + |ry| + 1) + \Phi\ (pass_1\ ry) - \Phi\ ry\\ &\le \log_2\ (|lx| + |ly| + 1) - \log_2\ (|lx| + |ry| + 1) + upperbound\ ry \qquad \text{by IH}\\ &\le upperbound\ hs \qquad \text{by } (*)\end{aligned}$$ □

Lemma 8.2 (Potential difference of *pass*$_1$)
$hs \neq \langle\rangle \Longrightarrow \Phi\ (pass_1\ hs) - \Phi\ hs \le 2 * \log_2\ |hs| - |hs|_R + 2$

Proof Show the following property by induction on the computation of *upperbound*

$$hs \neq \langle\rangle \Longrightarrow upperbound\ hs \le 2 * \log_2\ |hs| - |hs|_R + 2$$

and apply Lemma 8.1. □

Now we turn to *pass*$_2$:

Lemma 8.3 (Potential difference of *pass*$_2$)
$hs \neq \langle\rangle \Longrightarrow \Phi\ (pass_2\ hs) - \Phi\ hs \le \log_2\ |hs|$

Proof by induction on *hs*. The base case is trivial. The induction step (where $hs = \langle lx,\ x,\ rx\rangle$) is trivial if $rx = \langle\rangle$. Assume $rx = \langle ly,\ y,\ ry\rangle$. Now we need one more property of *pass*$_2$:

$$\exists la\ a.\ pass_2\ \langle ly, y, ry\rangle = \langle la, a, \langle\rangle\rangle$$

The proof is a straightforward induction on *ry*. Together with $|pass_2\ rx| = |rx|$ this implies $|la| + 1 = |rx|$. Together with $pass_2\ rx = \langle la, a, \langle\rangle\rangle$ this implies (by definition of the functions involved) that

$$\begin{aligned}&\Phi\ (link\ \langle lx, x, pass_2\ rx\rangle) - \Phi\ lx - \Phi\ (pass_2\ rx)\\ &= \log_2\ (|lx| + |rx| + 1) + \log_2\ (|lx| + |rx|) - \log_2\ |rx| \qquad (**)\end{aligned}$$

Thus the overall claim follows:

$$\begin{aligned}
&\Phi\ (pass_2\ hs) - \Phi\ hs \\
&= \Phi\ (link\ \langle lx, x, pass_2\ rx\rangle) - \Phi\ lx - \Phi\ rx - \log_2\ (|lx| + |rx| + 1) \\
&= \Phi\ (pass_2\ rx) - \Phi\ rx + \log_2\ (|lx| + |rx|) - \log_2\ |rx| && \text{by } (**) \\
&\le \log_2\ (|lx| + |rx|) && \text{by IH} \\
&\le \log_2\ |hs| && \square
\end{aligned}$$

We can also prove a logarithmic bound for the amortized complexity of *merge*:

Lemma 8.4 *If* $h_1 = \langle lx, x, \langle\rangle\rangle$ *and* $h_2 = \langle ly, y, \langle\rangle\rangle$ *then*

$$\Phi\ (merge\ h_1\ h_2) - \Phi\ h_1 - \Phi\ h_2 \le \log_2\ (|h_1| + |h_2|) + 1$$

Proof From

$$\begin{aligned}
&\Phi\ (merge\ h_1\ h_2) \\
&= \Phi\ (link\ \langle lx, x, h_2\rangle) \\
&= \Phi\ lx + \Phi\ ly + \log_2\ (|lx| + |ly| + 1) + \log_2\ (|lx| + |ly| + 2) \\
&= \Phi\ lx + \Phi\ ly + \log_2\ (|lx| + |ly| + 1) + \log_2\ (|h_1| + |h_2|)
\end{aligned}$$

it follows that

$$\begin{aligned}
&\Phi\ (merge\ h_1\ h_2) - \Phi\ h_1 - \Phi\ h_2 \\
&= \log_2\ (|lx| + |ly| + 1) + \log_2\ (|h_1| + |h_2|) - \log_2\ (|lx| + 1) - \log_2\ (|ly| + 1) \\
&\le \log_2\ (|h_1| + |h_2|) + 1 \\
&\qquad \text{because } \log_2\ (1 + x + y) \le 1 + \log_2\ (1 + x) + \log_2\ (1 + y) \text{ if } x,y \ge 0 \quad \square
\end{aligned}$$

Now we integrate the analysis into our framework from Section 3. We reuse the data type *op* from the section about skew heaps:

$exec\ Empty\ [] = \langle\rangle$
$exec\ Del\_min\ [h] = del\_min\ h$
$exec\ (Insert\ x)\ [h] = insert\ x\ h$
$exec\ Merge\ [h_1, h_2] = merge\ h_1\ h_2$

Running times are equated with the number of function calls, where non-recursive calls are lumped together:

$t_{pass1}\ \langle\rangle = 1$
$t_{pass1}\ \langle\_, \_, \langle\rangle\rangle = 1$
$t_{pass1}\ \langle\_, \_, \langle\_, \_, ry\rangle\rangle = t_{pass1}\ ry + 1$

$t_{pass2}\ \langle\rangle = 1$
$t_{pass2}\ \langle\_, \_, rx\rangle = t_{pass2}\ rx + 1$

$cost\ Empty\ [] = 1$
$cost\ Del\_min\ [\langle\rangle] = 1$
$cost\ Del\_min\ [\langle lx, \_, \_\rangle] = t_{pass2}\ (pass_1\ lx) + t_{pass1}\ lx$
$cost\ (Insert\ \_)\ \_ = 1$
$cost\ Merge\ \_ = 1$

The upper bounds for the amortized complexities:

$U\ Empty\ [] = 1$
$U\ (Insert\ \_)\ [h] = \log_2\ (|h| + 1) + 1$
$U\ Del\_min\ [h] = 3 * \log_2\ (|h| + 1) + 4$
$U\ Merge\ [h_1, h_2] = \log_2\ (|h_1| + |h_2| + 1) + 2$

The upper bound for *Insert* follows directly from (6) because $cost\ (Insert\ \_)\ \_ = 1$. The upper bound for *Del_min* follows from Lemma 8.2 and Lemma 8.3 by this easy inductive timing lemma:

$$t_{pass2}\ (pass_1\ lx) + t_{pass1}\ lx \leq |lx|_R + 2$$

The upper bound for *Merge* follows from Lemma 8.4; the cases where one of the heaps is empty are trivial.

8.2 Pairing Heaps via Lists

The above formulation of pairing heaps is not as readable as could be because everything is a tree. We have already indicated that some trees should be viewed as lists. Therefore Okasaki [38] represents pairing heaps as follows (in Isabelle notation):

datatype $'a\ heap = Empty \mid Hp\ 'a\ ('a\ heap\ list)$

This representation comes with the invariant that *Empty* only occurs at the root. The functional correctness proof does not require the invariant, but the amortized analysis seems to, to avoid $\log 0$; alternatively one may be able to add 1 to certain log-arguments, but that also complicates the proofs. Therefore we tried a third alternative that is free of structural invariants (of course one still needs the heap invariant for functional correctness):

datatype $'a\ hp = Hp\ 'a\ ('a\ hp\ list)$
type_synonym $'a\ heap = 'a\ hp\ option$

Although the proof text for this third alternative is slightly shorter than for the original tree version, it becomes conceptually less appealing: because three types are involved now ($'a\ hp$, $'a\ hp\ list$ and $'a\ heap$), we need three potential and three size functions. In contrast, the uniform binary tree representation also leads to uniform proofs.

As a final alternative we proved the complexity of Okasaki's version as a consequence of the complexity of the tree-based version. This proof relies on the following general result. We assume we are given

- an amortized analysis involving *exec*, *cost* and U on type $'s$,
- a second implementation of the same set of operations on type $'t$ involving $exec'$, $cost'$ and (claimed) upper bounds U'.
- a homomorphism *hom* from $'t$ to $'s$ such that $hom\ (exec'\ f\ ts) = exec\ f\ (map\ hom\ ts)$, $cost'\ f\ ts = cost\ f\ (map\ hom\ ts)$ and $U'\ f\ ts = U\ f\ (map\ hom\ ts)$.

Then U' states correct upper bounds on the amortized complexities of the second implementation. Okasaki's implementation is related back to the tree-based version as follows:

$hom\ Empty = \langle\rangle$
$hom\ (Hp\ x\ hs) = \langle homs\ hs, x, \langle\rangle\rangle$

$homs\ [] = \langle\rangle$
$homs\ (Hp\ x\ lhs \cdot rhs) = \langle homs\ lhs, x, homs\ rhs\rangle$

The required proofs are straightforward inductions.

This subsection has concentrated on the essence. For the details the reader should consult the complete theories available online [29,34].

8.3 Tighter bounds

Pairing Heaps have been designed in order to have the same (optimal) amortized running times as Fibonacci Heaps, but without their big overhead. In the original paper [13] the authors proved the logarithmic bounds verified above. Iacono [22] proved that *insert* and *merge* do indeed have constant amortized complexity.

The picture becomes more complicated when *decrease key* is also supported, an operation that is important in practice because it is used in Dijkstra's algorithm. However, efficient implementations of *decrease key* rely on references. It turns out that the precise amortized complexity of *decrease key* is still open.

A recent new variant of pairing heaps with perfect asymptotic running times are *rank-pairing heaps* [14]. The introduction to that article also provides a good overview of the state of the art in pairing heaps and their complexity.

9 Conclusion

This paper has presented a framework for amortized analysis and detailed, completely algebraic proofs of amortized complexity of a range of nontrivial data structures. Apart from the design of the framework, the main challenges were finding the algebraic proofs and verifying them in Isabelle. For skew heaps, splay trees and splay heaps, algebraic proofs had already been given in the literature. For pairing heaps we found that recasting the proof of Theorem 1 in [13] into purely algebraic form was non-trivial. The verification in Isabelle was of average difficulty, except for the challenging nonlinear arithmetic in Section 6.3. Better automation of arithmetic is needed to support algorithm analyses in theorem provers.

The following table shows the sizes (in number of lines) of the Isabelle theories containing the amortized analyses of the four data structures we examined:

Skew heap	Splay tree	Splay heap	Pairing heap
200	560 (640)	220	260

For splay trees, the number in parentheses refers to the theory for the optimal analysis from Section 6.3. As is typical in mathematics, the size of the final proofs belies the effort that went into constructing them. At the same time it is encouraging that the formal proofs are of a size comparable to carefully written pen-and-paper proofs.

Although the focus of our analyses are functional programs, one can also view these programs as models or abstractions. The dynamic table in Section 4.2 is a case in point: it is not a useful functional data structure but a model of a dynamic array where the indexing operations and the contents have been omitted because they are irrelevant for the analysis. In general, the amortized analysis depends only on the input-output behaviour and the complexity of each operation, given by *exec* and *cost*. Hence the analysis is valid for any implementation that behaves like *exec* and *cost*, regardless of the implementation language. In fact, the standard imperative implementations of all of our examples have the same complexity as their functional counterpart.

Acknowledgements Berry Schoenmakers patiently answered many questions about his work. We thank the referees for their careful reading and helpful suggestions.

References

1. R. Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2), 2011.
2. R. Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.
3. H. Brinkop and T. Nipkow. Pairing heap. *Archive of Formal Proofs*, 2016. `http://isa-afp.org/entries/Pairing_Heap.html`, Formal proof development.
4. Q. Carbonneaux, J. Hoffmann, T. W. Reps, and Z. Shao. Automated resource analysis with Coq proof objects. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification, CAV 2017, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 64–85. Springer, 2017.
5. A. Charguéraud and F. Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *J. Automated Reasoning*. To appear.
6. A. Charguéraud and F. Pottier. Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 137–153. Springer, 2015.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
8. K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th Symposium on Principles of Programming Languages*, POPL '00, pages 184–198. ACM, 2000.
9. N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proc. 35th Symposium on Principles of Programming Languages*, POPL '08, pages 133–144. ACM, 2008.
10. N. Danner, D. R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proc. International Conference on Functional Programming*, ICFP 2015, pages 140–151. ACM, 2015.
11. N. Danner, J. Paykin, and J. Royer. A static cost analysis for a higher-order language. In *Proc. Workshop Programming Languages Meets Program Verification*, PLPV '13, pages 25–34. ACM, 2013.
12. P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science*, 79(1):37 – 109, 1991.
13. M. L. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
14. B. Haeupler, S. Sen, and R. E. Tarjan. Rank-pairing heaps. *SIAM J. Comput.*, 40(6):1463–1485, 2011.
15. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.
16. J. Harrison. Verifying nonlinear real formulas via sums of squares. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 102–118. Springer, 2007.
17. T. Hickey and J. Cohen. Automating program analysis. *J. ACM*, 35(1):185–220, 1988.
18. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.
19. J. Hoffmann, A. Das, , and S.-C. Weng. Towards automatic resource bound analysis for OCaml. In *Proc. 44th Symposium on Principles of Programming Languages*, POPL '17, pages 359–373. ACM, 2017.
20. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symposium Principles of Programming Languages*, pages 185–197, 2003.
21. L. Hupel and T. Nipkow. A verified compiler from Isabelle/HOL to CakeML. In A. Ahmed, editor, *European Symposium on Programming (ESOP 2018)*, volume ? of *LNCS*, pages ?–? Springer, 2018.
22. J. Iacono. Improved upper bounds for pairing heaps. In M. M. Halldórsson, editor, *Algorithm Theory - SWAT 2000*, volume 1851 of *LNCS*, pages 32–45. Springer, 2000.
23. A. Kaldewaij and B. Schoenmakers. The derivation of a tighter bound for top-down skew heaps. *Information Processing Letters*, 37:265–271, 1991.
24. A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNCS*, pages 589–603. Springer, 2006.

25. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Symp. Principles of Programming Languages, POPL '14*, pages 179–191. ACM, 2014.
26. D. Le Métayer. ACE: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.
27. R. Madhavan, S. Kulal, and V. Kuncak. Contract-based resource verification for higher-order functions with memoization. In *Principles of Programming Languages (POPL)*, 2017.
28. J. A. McCarthy, B. Fetscher, M. S. New, D. Feltey, and R. B. Findler. A Coq library for internal verification of running-times. In O. Kiselyov and A. King, editors, *Functional and Logic Programming (FLOPS 2016)*, volume 9613 of *LNCS*, pages 144–162. Springer, 2016.
29. T. Nipkow. Amortized complexity verified. *Archive of Formal Proofs*, 2014. `http://isa-afp.org/entries/Amortized_Complexity.shtml`, Formal proof development.
30. T. Nipkow. Skew heap. *Archive of Formal Proofs*, 2014. `http://isa-afp.org/entries/Skew_Heap.shtml`, Formal proof development.
31. T. Nipkow. Splay tree. *Archive of Formal Proofs*, 2014. `http://isa-afp.org/entries/Splay_Tree.shtml`, Formal proof development.
32. T. Nipkow. Amortized complexity verified. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pages 310–324. Springer, 2015.
33. T. Nipkow. Automatic functional correctness proofs for functional search trees. In J. Blanchette and S. Merz, editors, *Interactive Theorem Proving (ITP 2016)*, LNCS. Springer, 2016.
34. T. Nipkow. Pairing heap. *Archive of Formal Proofs*, 2016. `http://isa-afp.org/entries/Pairing_Heap.shtml`, Formal proof development.
35. T. Nipkow. Verified root-balanced trees. In B.-Y. E. Chang, editor, *Asian Symposium on Programming Languages and Systems, APLAS 2017*, volume 10695 of *LNCS*, pages 255–272. Springer, 2017.
36. T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL.* Springer, 2014. `http://concrete-semantics.org`.
37. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
38. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
39. D. Sands. Complexity analysis for a lazy higher-order language. In N. Jones, editor, *European Symposium on Programming (ESOP)*, volume 432 of *LNCS*, pages 361–376. Springer, 1990.
40. B. Schoenmakers. A systematic analysis of splaying. *Information Processing Letters*, 45:41–50, 1993.
41. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
42. D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, 1986.
43. R. E. Tarjan. Amortized complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, 1985.
44. D. Traytel, S. Berghofer, and T. Nipkow. Extending Hindley-Milner type inference with coercive structural subtyping. In H. Yang, editor, *APLAS 2011*, volume 7078 of *LNCS*, pages 89–104. Springer, 2011.
45. P. B. Vasconcelos and K. Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In P. Trinder, G. Michaelson, and R. Pena, editors, *Implementation of Functional Languages, IFL 2003*, volume 3145 of *LNCS*, pages 86–101. Springer, 2004.
46. B. Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.
47. M. Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. `http://mediatum.ub.tum.de?id=601724`.