Verified Root-Balanced Trees

Tobias Nipkow*

Technische Universität München
http://www.in.tum.de/~nipkow

Abstract. Andersson introduced *general balanced trees*, search trees based on the design principle of partial rebuilding: perform update operations naively until the tree becomes too unbalanced, at which point a whole subtree is rebalanced. We define and analyze a functional version of general balanced trees which we call *root-balanced trees*. Using a lightweight model of execution time, amortized logarithmic complexity is verified in the theorem prover Isabelle. Experimental results show competitiveness of root-balanced with AVL and red-black trees.

1 Introduction

An old idea from the search tree literature is *partial rebuilding* [26]. Search trees are not rebalanced during every update operation but are allowed to degenerate before they are eventually rebalanced in a more drastic manner: a whole subtree is rebalanced to optimal height (in linear time). We build on the work of Andersson [2] who rebalances only if the height of the tree is no longer logarithmically bounded by the size of the tree. We call these trees *root-balanced*. We recast Andersson's ideas in a functional framework and verify their correctness in Isabelle: the *amortized* complexity is logarithmic. The main contributions are:

- The (as far as we know) first published executable formulation of root-balanced trees, imperative or functional. It is expressed in the functional language of the theorem prover Isabelle/HOL [24,23].
- A lightweight approach to modelling execution time of functional programs.
- The first formal verification of the amortized complexity of root-balanced trees. We develop the code by refinement and present the main definitions and theorems for the complexity proof. The full Isabelle definitions and proofs are found in the online Archive of Formal Proofs [22].
- Logarithm-free code. A direct implementation of Andersson's algorithms needs to compute logarithms. Because floating point computations are inexact this might lead to different behaviour and different actual complexity.
- The first published empirical evaluation of root-balanced trees. It shows that they are competitive with AVL and red-black trees.

Note that in a functional setting, amortized complexity reasoning is only valid if the data structure under consideration is used in a single-threaded manner, unless one employs lazy evaluation or memoization [25], which we do not.

* Supported by DFG Koselleck grant NI 491/16-1

We do not discuss the functional correctness proofs because they follow a method described elsewhere [21] and are automatic. The challenge is not correctness but amortized complexity.

2 Basics

Type variables are denoted by $'a$, $'b$, etc. The notation $t :: \tau$ means that term t has type τ. Type constructors follow postfix syntax, eg $'a\ set$ is the type of sets of elements of type $'a$.

The types *nat*, *int* and *real* represent the sets $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$. In this paper we often drop the coercion functions that are embeddings (like $real :: nat \to real$) but show $nat :: int \to nat$ because it maps the negative numbers to 0.

Lists over type $'a$, type $'a\ list$, come with the empty list [], the infix constructor ".", the infix append operator @, *hd* (head), *tl* (tail), and the enumeration syntax $[x_1, \ldots, x_n]$.

Note that "=" on type *bool* means "$\longleftrightarrow$".

2.1 Trees

datatype $'a\ tree = \langle\rangle \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

We abbreviate $Node\ l\ a\ r$ by $\langle l, a, r\rangle$. The size ($|t|$) and height ($height\ t$) of a tree are defined as usual, starting with 0 for leaves. In addition we define $|t|_1 = |t| + 1$. Function *inorder* is defined canonically. The minimal height is defined as follows:

$$min\_height\ \langle\rangle = 0$$
$$min\_height\ \langle l, \_, r\rangle = min\ (min\_height\ l)\ (min\_height\ r) + 1$$

We call a tree *balanced* iff its height and minimal height differ by at most 1:

$$balanced\ t = (height\ t - min\_height\ t \leq 1)$$

3 Balancing Trees

A number of imperative algorithms have been published that balance trees in linear time (eg [11,29]). We present a linear functional algorithm that first projects the tree to a list in linear time (using accumulation)

$$inorder_2\ \langle\rangle\ xs = xs$$
$$inorder_2\ \langle l, x, r\rangle\ xs = inorder_2\ l\ (x \cdot inorder_2\ r\ xs)$$

and builds the balanced tree from that:

$$\begin{array}{l} bal\ n\ xs = \\ (\mathsf{if}\ n = 0\ \mathsf{then}\ (\langle\rangle, xs) \\ \ \mathsf{else\ let}\ m = n\ div\ 2; \\ \qquad\quad (l, ys) = bal\ m\ xs; \\ \qquad\quad (r, y) = bal\ (n - 1 - m)\ (tl\ ys) \\ \qquad \mathsf{in}\ (\langle l, hd\ ys, r\rangle, y)) \end{array}$$

$bal\_list\ n\ xs = (\mathsf{let}\ (t,\ ys) = bal\ n\ xs\ \mathsf{in}\ t)$

$bal\_tree\ n\ t = bal\_list\ n\ (inorder_2\ t\ [])$

$balance\_tree\ t = bal\_tree\ |t|\ t$

This algorithm is most likely not new but we need it and its properties for rebalancing subtrees.

Lemma 1 *The order of elements is preserved and the result is balanced:*

$n \leq |t| \longrightarrow inorder\ (bal\_tree\ n\ t) = take\ n\ (inorder\ t)$
$balanced\ (bal\_tree\ n\ t)$

In order to avoid confusion with other notions of balancedness we refer to the above notion as *optimally balanced* in the text.

4 Time

There have been a number of proposals in the literature on how to model and analyze execution time of functional programs within functional programs or theorem provers (see Section 9). The key techniques are type systems and monads. We will also make use of a resource monad to accumulate execution time. The result is a lightweight approach to modelling and analyzing call-by-value execution time.

The basic principle of our approach is that the users define their programs in the monad and derive two separate functions from it, one that performs the computation and one that yields the time. Our time monad is based on

datatype $'a\ tm = TM\ 'a\ nat$

which combines a value with an internal clock:

$val\ (TM\ v\ n) = v$
$time\ (TM\ v\ n) = n$

The standard monadic combinators are

$$\begin{aligned} s \gg\!= f = (&\mathsf{case}\ s\ \mathsf{of} \\ &TM\ u\ m \Rightarrow \mathsf{case}\ f\ u\ \mathsf{of} \\ &\qquad TM\ v\ n \Rightarrow TM\ v\ (m + n)) \\ return\ v = &\ TM\ v\ 0 \end{aligned}$$

where $\gg\!=$ (*bind*) adds the clocks of two computations and *return* is for free. Below we employ the usual do-notation instead of *bind*.

For simplicity our clock counts only function calls. In order to charge one clock tick for a function call we define the infix operator $=_1$ that does just that:

$lhs =_1 rhs$ abbreviates $lhs = (rhs \gg\!= tick)$ where $tick\ v = TM\ v\ 1$.

That is, when defining a function f whose time we want to measure we need to write every defining equation $f\ p = t$ as $f\ p =_1 t$.

It is up to the users in how much detail they model the execution time, that is, how much of a computation they embed in the monad. In this paper we count all function calls (via the monad) except for constructors and functions on booleans and numbers. It would be easy to lift all of the latter functions into the monad as well, say with unit costs; this would merely lead to bigger constants in applications. This means that the concrete constants that appear in the time formulas we will prove are an underapproximation but could be made more precise if desired.

Let us look at an example, the monadic definition of function $inorder_2\_tm$:

$$
\begin{array}{l}
inorder_2\_tm\ \langle\rangle\ xs =_1 return\ xs \\
inorder_2\_tm\ \langle l,\ x,\ r\rangle\ xs =_1 do\ \{ \\
\quad rs \leftarrow inorder_2\_tm\ r\ xs; \\
\quad inorder_2\_tm\ l\ (x \cdot rs) \\
\}
\end{array}
$$

From every monadic function f_tm we define the projections f and t_f on the value and the time. For $inorder_2\_tm$ these are

$$
\begin{array}{l}
inorder_2\ t\ xs = val\ (inorder_2\_tm\ t\ xs) \\
t\_inorder_2\ t\ xs = time\ (inorder_2\_tm\ t\ xs)
\end{array}
$$

From these definitions we prove (automatically, by simplification) recursion equations that follow the recursion of the monadic version:

$$
\begin{array}{l}
inorder_2\ \langle\rangle\ xs = xs \\
inorder_2\ \langle l,\ x,\ r\rangle\ xs = (\mathsf{let}\ rs = inorder_2\ r\ xs\ \mathsf{in}\ inorder_2\ l\ (x \cdot rs))
\end{array}
$$

$$
\begin{array}{l}
t\_inorder_2\ \langle\rangle\ xs = 1 \\
t\_inorder_2\ \langle l,\ x,\ r\rangle\ xs = \\
t\_inorder_2\ r\ xs + t\_inorder_2\ l\ (x \cdot inorder_2\ r\ xs) + 1
\end{array}
$$

These, rather than the monadic versions, are used in the rest of the verification. For presentation reasons we sometimes expand lets if the let-bound variable occurs exactly once in the body, as in the definition of $inorder_2$ in Section 3.

For the running time function it is often possible to prove some non-recursive bound or even an exact value, for example

$$t\_inorder_2\ t\ xs = 2 * |t| + 1$$

The step from f_tm to f needs some attention because it must not change the running time in the process: f is the actual code, f_tm is only a means for describing the computation of the value and its time complexity simultaneously. For example, $f\_tm\ x = do\ \{\ y \leftarrow g\ x;\ return\ (h\ y\ y)\}$ should be transformed into $f\ x = (\mathsf{let}\ y = g\ x\ \mathsf{in}\ h\ y\ y)$, not into $f\ x = h\ (g\ x)\ (g\ x)$ because the latter evaluates $g\ x$ twice. Of course we cannot prove this because HOL functions are extensional and have no running time complexity. We can only argue that an

intensional interpretation of the derived equation for f has the running time ascribed to it by the counter in the definition of *f_tm*. Our argument is that in the derivation of the equations for f we use only rewriting with the definitions of *f_tm* and f and with the following rules for pushing *val* inside and eliminating the clock.

$$val\ (return\ x) = x$$
$$val\ (m \gg\!= f) = (\mathsf{let}\ x = val\ m\ \mathsf{in}\ val\ (f\ x))$$
$$val\ (tick\ x) = x$$
$$val\ (\mathsf{let}\ x = t\ \mathsf{in}\ f\ x) = (\mathsf{let}\ x = t\ \mathsf{in}\ val\ (f\ x))$$
$$val\ (\mathsf{if}\ c\ \mathsf{then}\ x\ \mathsf{else}\ y) = (\mathsf{if}\ c\ \mathsf{then}\ val\ x\ \mathsf{else}\ val\ y)$$

There are also obvious rules for case-expressions. All these rules do not change the running time of the intensional interpretation of the terms. The rules for *val* do not duplicate or erase parameters.

Outside of this section we do not show the original monadic definition of functions *f_tm* but only the derived equations for f; the definition of *t_f* is not shown either, it can be inferred from *f*.

Below we need the following results about the running times of the functions *size_tree* (normally shown as $|.|$) and *bal_tree*:

Lemma 2 *t_size_tree* $t = 2 * |t| + 1$ *and t_bal_tree* $|xs|\ xs = 4 * |xs| + 3$

Note that the soundness of our approach depends on the users observing data abstraction: monadic definitions must only use the monadic combinators $\gg\!=$ and *return* (and $=_1$ or *tick*) but must not take monadic values apart, e.g. by pattern matching on *TM*. Alternatively one can write a tool that takes a normal function definition for f and generates the definition for *t_f* from it.

5 Root-Balanced Trees: Insertion

Root-balanced trees are (binary) search trees. To express that the elements in the trees must be ordered we employ Isabelle's type classes [19,32]. In the rest of the paper we assume that the type variable $'a$ in $'a\ tree$ is of class *linorder*, i.e., there are predicates $\leq$ and $<$ on $'a$ that form a linear order. Instead of using $\leq$ and $<$ directly we define a 3-way comparison function *cmp*:

datatype $cmp\_val = LT \mid EQ \mid GT$
$$cmp\ x\ y = (\mathsf{if}\ x < y\ \mathsf{then}\ LT\ \mathsf{else}\ \mathsf{if}\ x = y\ \mathsf{then}\ EQ\ \mathsf{else}\ GT)$$

Root-balanced trees should satisfy some minimal balance criterion at the root, something like $height\ t \leq c * \log_2\ |t|$. We make this minimal balance criterion a parameter of our development: $bal_i :: nat \to nat \to bool$ where the subscript stands for "insertion". The two arguments of bal_i are the size and the height of the tree. The parameter bal_i is subject to two assumptions:

Assumption 1 (Monotonicity) $bal_i\ n\ h \wedge n \leq n' \wedge h' \leq h \longrightarrow bal_i\ n'\ h'$

Assumption 2 $bal_i\ |t|\ (height\ (balance\_tree\ t))$

Insertion works as follows. As for ordinary search trees, it searches for the element that is to be inserted. If that search ends up at a leaf, that leaf is replaced by a new singleton node. Then it checks if the new node has unbalanced the tree at the root (because it became too high). If so, then on the way back up it checks at every subtree if that is balanced, until it finds a subtree that is unbalanced and which then gets rebalanced. Thus the algorithm has to distinguish between unbalanced, balanced and unchanged trees (in case the element to be inserted was already in the tree):

datatype $'a\ up = Same \mid Bal\ ('a\ tree) \mid Unbal\ ('a\ tree)$

That is, if (a recursive call of) insertion returns *Same*, the tree remains unchanged; if it returns *Bal t*, the new subtree is t and no (more) rebalancing is necessary; if it returns *Unbal t*, the new subtree is t and some subtree higher up needs to be rebalanced because the root has become unbalanced.

5.1 A Naive Implementation

To reduce the complexity of the verification we start with a first inefficient implementation and show that it is functionally correct. A second efficient implementation will then be shown to be functionally equivalent and to have the desired complexity.

Function $ins :: nat \to nat \to 'a \to 'a\ tree \to 'a\ up$ inserts an element:

$$
\begin{array}{l}
ins\ n\ d\ x\ \langle\rangle = \\
(\mathsf{if}\ bal_i\ (n+1)\ (d+1)\ \mathsf{then}\ Bal\ \langle\langle\rangle, x, \langle\rangle\rangle\ \mathsf{else}\ Unbal\ \langle\langle\rangle, x, \langle\rangle\rangle) \\
ins\ n\ d\ x\ \langle l, y, r\rangle = (\mathsf{case}\ cmp\ x\ y\ \mathsf{of} \\
\qquad\qquad\qquad\qquad LT \Rightarrow up\ y\ r\ False\ (ins\ n\ (d+1)\ x\ l) \\
\qquad\qquad\qquad \mid EQ \Rightarrow Same \\
\qquad\qquad\qquad \mid GT \Rightarrow up\ y\ l\ True\ (ins\ n\ (d+1)\ x\ r))
\end{array}
$$

Parameter n is the size of the whole tree, parameter d the depth of the recursion, i.e. the distance from the root. Both parameters are needed in order to decide at the leaf level if the tree has become unbalanced because that information is needed on the way back up. Checking and rebalancing is performed by function *up* below, where *sib* is the sibling of the subtree inside u, *twist* indicates whether it is the left or right sibling and x is the contents of the node:

$$
\begin{array}{l}
up\ x\ sib\ twist\ u = \\
(\mathsf{case}\ u\ \mathsf{of} \\
\quad Same \Rightarrow Same \\
\mid Bal\ t \Rightarrow Bal\ (node\ twist\ t\ x\ sib) \\
\mid Unbal\ t \Rightarrow \\
\quad \mathsf{let}\ t' = node\ twist\ t\ x\ sib; \\
\qquad h' = height\ t'; \\
\qquad n' = |t'| \\
\quad \mathsf{in}\ \mathsf{if}\ bal_i\ n'\ h'\ \mathsf{then}\ Unbal\ t'\ \mathsf{else}\ Bal\ (balance\_tree\ t'))
\end{array}
$$

$node\ twist\ s\ x\ t = (\mathsf{if}\ twist\ \mathsf{then}\ \langle t,\ x,\ s\rangle\ \mathsf{else}\ \langle s,\ x,\ t\rangle)$

Obviously *ins* increases the size by 1 (if the element is not there already); the height changes as follows:

$ins\ n\ d\ x\ t = Bal\ t' \longrightarrow height\ t' \leq height\ t + 1$
$ins\ n\ d\ x\ t = Unbal\ t' \longrightarrow height\ t \leq height\ t' \leq height\ t + 1$

In the first case the height may actually shrink due to rebalancing. The proof is by simultaneous induction and relies on Assumption 1 and 2.

The return value *Unbal* signals that the tree has become unbalanced at the root. Formally:

Lemma 3 $ins\ n\ d\ x\ t = Unbal\ t' \longrightarrow \neg\ bal_i\ (n + 1)\ (height\ t' + d)$

The proof is by induction and uses monotonicity of bal_i. An easy consequence:

Lemma 4 $ins\ n\ (d + 1)\ x\ l = Unbal\ l' \wedge bal_i\ n\ (height\ \langle l,\ y,\ r\rangle + d) \longrightarrow height\ r < height\ l'$

There is a symmetric lemma for r instead of l. These two lemmas tell us that if insertion unbalances a balanced tree, then it climbs back up what has become the longest path in the tree.

The top-level insertion function is *insert*:

$$\begin{array}{l} insert\ x\ t = (\mathsf{case}\ ins\ |t|\ 0\ x\ t\ \mathsf{of} \\ \qquad\qquad Same \Rightarrow t \\ \qquad\quad \mid Bal\ t' \Rightarrow t') \end{array}$$

Note that the call of *ins* in *insert* cannot return *Unbal* because (by definition of *up*) this only happens if the tree is balanced, which contradicts Lemma 3:

Lemma 5 $|t| \leq n \longrightarrow ins\ n\ 0\ a\ t \neq Unbal\ t'$

Hence proofs about *insert* do not need to consider case *Unbal*.

5.2 An Efficient Implementation

The above implementation computes the sizes and heights of subtrees explicitly and repeatedly as it goes back up the tree. We will now perform that computation incrementally. Of course one can store that information in each node but the beauty of this search tree is that only at the very root we need to store some extra information, the size of the tree. The incremental version of the algorithm works with a modified data type $'a\ up2$

datatype $'a\ up2 = Same_2 \mid Bal_2\ ('a\ tree) \mid Unbal_2\ ('a\ tree)\ nat\ nat$

where $Unbal_2\ t\ n\ h$ passes n and h, the size and height of t, back up the tree. The new version of function *up* is up_2:

$up_2\ x\ sib\ twist\ u =$
(case u of
  $Same_2 \Rightarrow Same_2$
| $Bal_2\ t \Rightarrow Bal_2\ (node\ twist\ t\ x\ sib)$
| $Unbal_2\ t\ n_1\ h_1 \Rightarrow$
  let $n_2 = |sib|$;
    $h_2 = height\ sib$;
    $t' = node\ twist\ t\ x\ sib$;
    $n' = n_1 + n_2 + 1$;
    $h' = max\ h_1\ h_2 + 1$
  in if $bal_i\ n'\ h'$ then $Unbal_2\ t'\ n'\ h'$ else $Bal_2\ (bal\_tree\ n'\ t')$)

Note that instead of *balance_tree* we call *bal_tree* because that avoids the computation of the size of the tree that we have already. There are also corresponding new versions of *ins* and *insert*:

$ins_2\ n\ d\ x\ \langle\rangle =$
(if $bal_i\ (n + 1)\ (d + 1)$ then $Bal_2\ \langle\langle\rangle, x, \langle\rangle\rangle$ else $Unbal_2\ \langle\langle\rangle, x, \langle\rangle\rangle\ 1\ 1$)
$ins_2\ n\ d\ x\ \langle l, y, r\rangle =$ (case $cmp\ x\ y$ of
  $LT \Rightarrow up_2\ y\ r\ False\ (ins_2\ n\ (d + 1)\ x\ l)$
| $EQ \Rightarrow Same_2$
| $GT \Rightarrow up_2\ y\ l\ True\ (ins_2\ n\ (d + 1)\ x\ r)$)

$insert_2\ x\ (t, n) =$ (case $ins_2\ n\ 0\ x\ t$ of
  $Same_2 \Rightarrow (t, n)$
| $Bal_2\ t' \Rightarrow (t', n + 1)$)

Note that the top-level function $insert_2$ operates on pairs (t, n) where $n = |t|$. The relationship between $ins/insert$ and $ins_2/insert_2$ is easy to state and prove:

$(ins_2\ n\ d\ x\ t = Same_2) = (ins\ n\ d\ x\ t = Same)$
$(ins_2\ n\ d\ x\ t = Bal_2\ t') = (ins\ n\ d\ x\ t = Bal\ t')$
$(ins_2\ n\ d\ x\ t = Unbal_2\ t'\ n'\ h') =$
$(ins\ n\ d\ x\ t = Unbal\ t' \wedge n' = |t'| \wedge h' = height\ t')$

$(insert_2\ x\ (t, |t|) = (t', n')) = (t' = insert\ x\ t \wedge n' = |t'|)$

Case $Unbal_2$ in up_2 is suboptimal because *sib* is traversed twice. But instead of introducing a combined size and height function it turns out we can simply drop the computation of *height sib*. The reason is that, if initially balanced, the tree becomes unbalanced only if we have inserted a new node at the end of a longest path, which means that the height is determined by that path alone. This is what Lemma 4 expresses. The final version up_3 is obtained from up_2 by replacing the right-hand side of case $Unbal_2$ with the following expression:

let $n_2 = |sib|$;
  $t' = node\ twist\ t\ x\ sib$;
  $n' = n_1 + n_2 + 1$;
  $h' = h_1 + 1$
in if $bal_i\ n'\ h'$ then $Unbal_2\ t'\ n'\ h'$ else $Bal_2\ (bal\_tree\ n'\ t')$

The corresponding insertion functions $ins_3/insert_3$ look like $ins_2/insert_2$ but call up_3/ins_3. Function $insert_3$ is the final implementation of insertion.

The relationship between level 2 and 3 requires that the tree is balanced and is more involved to prove:

$$bal_i\ n\ (height\ t + d) \longrightarrow ins_3\ n\ d\ x\ t = ins_2\ n\ d\ x\ t$$
$$bal_i\ n\ (height\ t) \longrightarrow insert_3\ x\ (t, n) = insert_2\ x\ (t, n)$$

The precondition is needed for using Lemma 4.

We will move silently between the three levels using the above equivalences.

5.3 Amortized Complexity

In the worst case, insertion can require a linear amount of work because the whole tree has to be rebalanced. We will show that each rebalancing must be preceded by a linear number of insertions without rebalancing over which the cost of the eventual rebalancing can be spread, increasing the cost of each insertion only by a constant. If bal_i is defined such that the height of balanced trees is logarithmic in the size, insertion has logarithmic amortized cost.

The core of the potential function argument by Andersson [2] is the *imbalance* of a node:

$$imbal\ \langle\rangle = 0$$
$$imbal\ \langle l, \_, r\rangle = nat\ |int\ |l| - int\ |r||\ -\ 1$$

Thus $imbal\ \langle l, \_, r\rangle$ is the absolute value of the difference in size of l and r, minus 1. Because the subtraction of 1 is at type *nat*, it is cut off at 0. A consequence of subtracting 1 (which Andersson [1] does not do) will be that optimally balanced trees have potential 0.

The key property of *imbal* is that insertion into a subtree can increase the imbalance of a node by at most 1: defining $\delta\ t\ s = real(imbal\ t) - real(imbal\ s)$ we obtain

Lemma 6 $ins\ n\ d\ x\ t = Bal\ t' \vee ins\ n\ d\ x\ t = Unbal\ t' \longrightarrow$
$\delta\ (node\ tw\ t'\ y\ s)\ (node\ tw\ t\ y\ s) \leq 1$

This follows by definition of *imbal* from the fact that the height of t' can have increased by at most one.

Now we add another assumption about bal_i: when we climb back up the tree after an insertion and find an unbalanced node whose higher child (where we must have come from) is balanced, then the imbalance of the node is proportional to its size:

Assumption 3 $\neg\ bal_i\ |t|\ (height\ t) \wedge bal_i\ |hchild\ t|\ (height\ (hchild\ t)) \wedge t \neq \langle\rangle \longrightarrow |t|_1 \leq e * (imbal\ t + 1)$

where $hchild\ \langle l, \_, r\rangle = (\mathsf{if}\ height\ l \leq height\ r\ \mathsf{then}\ r\ \mathsf{else}\ l)$ and e is some real number greater 0.

We define the actual potential function Φ as the sum of all imbalances in a tree, scaled by $6 * e$:

$\Phi\ \langle\rangle = 0$
$\Phi\ \langle l,\ x,\ r\rangle = 6 * e * \mathit{imbal}\ \langle l,\ x,\ r\rangle + \Phi\ l + \Phi\ r$

The factor 6 comes from the complexities of the size computations and the rebalancing. Both are linear, but with factors 2 and 4 (Lemma 2). These linear complexities need to be paid for by the potential.

Clearly $0 \leq \Phi\ t$, as is required of a potential function. Moreover, the potential of a balanced tree is 0:

Lemma 7 $\Phi\ (\mathit{balance\_tree}\ t) = 0$

The main theorem expresses that the amortized complexity of insert_3 is linear in the height of the tree.

Theorem 1 $\mathit{bal}_i\ |t|\ (\mathit{height}\ t) \wedge \mathit{insert}_3\ a\ (t,\ |t|) = (t',\ n') \longrightarrow$
$\mathit{t\_insert}_3\ a\ (t,\ |t|) + \Phi\ t' - \Phi\ t \leq (6 * e + 2) * (\mathit{height}\ t + 1) + 1$

Now we plug that result into a framework for amortized analysis, and finally we show that for certain interpretations of bal_i and e, the height of the tree is logarithmic in the size.

Instantiating the Framework We use an existing framework [20] for the analysis of the amortized complexity of data structures. It guarantees that, given certain key theorems, the data structure indeed has the claimed complexity. We instantiate the framework as follows: The state space consists of pairs $(t,\ n)$. The initial state is $(\langle\rangle,\ 0)$. The invariant is $\lambda(t,\ n).\ n = |t| \wedge \mathit{bal}_i\ |t|\ (\mathit{height}\ t)$; the invariance proof relies on asumptions 1 and 2. The potential function is $\lambda(t,\ n).\ \Phi\ t$. The amortized complexity of $\mathit{insert}_3\ x\ (t,\ n)$ is bounded from above by $(6 * e + 2) * (\mathit{height}\ t + 1) + 1$ (Theorem 1).

Logarithmic Height So far the verification was parameterized by bal_i and e subject to some assumptions. Now we give a concrete instantiation that guarantees a logarithmic bound on the height. We follow Andersson [2] and define

$$\mathit{bal}_i\ n\ h\ =\ h \leq \lceil c * \log_2\ (n + 1)\rceil \tag{1}$$

for some arbitrary $c > 1$.

We have to show that all assumptions are satisfied. Assumption 1 (Monotonicity) clearly holds. Assumption 2 is a consequence of the lemma $\mathit{height}\ (\mathit{balance\_tree}\ t) = \mathit{nat}\ \lceil\log_2\ (|t| + 1)\rceil$. Assumption 3 follows by setting

$$e = 2^{1/c}/(2 - 2^{1/c}) \tag{2}$$

Thus we know that (1) and (2) satisfy the above parameterized complexity analysis. Because we proved that bal_i is an invariant, we know that the height of the tree is bounded from above by $\lceil c * \log_2\ |t|_1\rceil$. Thus the amortized complexity of insertion is bounded from above by $(6 * e + 2) * (\lceil c * \log_2\ |t|_1\rceil + 1) + 1$.

6 Root-Balanced Trees: Deletion

The key idea is to perform standard deletions (no balancing) until enough deletions have occurred to pay for rebalancing at the *root*. This means that the data structure needs to maintain a counter of the number of deletions; the counter is reset when the root is rebalanced because of a deletion. Because balancing is linear, any fixed fraction of the size of the tree will work. We parameterize the whole development by that fraction, a constant $c_d > 0$. Thus the balance test $bal_d :: nat \to nat \to bool$ to be used after each deletion is defined as

$$bal_d\ n\ dl = (dl < c_d * (n + 1))$$

where n is the number of nodes in the tree and dl the number of deletions that have occurred since the last rebalancing after a deletion.

We extend the development of the previous section with a deletion function and a new top-level insertion function. Many of the existing functions and lemmas are reused in the extended setting.

6.1 A Naive Implementation

The main supporting lemmas are proved about an implementation where the size of the tree is not cached. That is rectified in a second step. The new top-level insertion function $insert_d$ operates on a pair of a tree and the deletion counter. We build upon function *ins* from Section 5.1.

$$\begin{array}{l} insert_d\ x\ (t,\ dl) = (\mathsf{case}\ ins\ (|t| + dl)\ 0\ x\ t\ \mathsf{of} \\ \qquad\qquad\qquad\qquad\quad Same \Rightarrow t \\ \qquad\qquad\qquad\qquad \mid Bal\ t' \Rightarrow t', \\ \qquad\qquad\qquad\qquad dl) \end{array}$$

Why is the deletion counter added to the size? That way the sum stays invariant under deletion and the invariant $bal_i\ (|t| + dl)\ (height\ t)$ for insertion will also remain invariant under deletion.

Deletion works as for unbalanced trees. The deletion function *del* returns *'a tree option* to signal whether the element was in the tree or not:

datatype *'a option* = *None* | *Some 'a*

```
del x ⟨⟩ = None
del x ⟨l, y, r⟩ =
(case cmp x y of
   LT ⇒ up_d y r False (del x l)
 | EQ ⇒ if r = ⟨⟩ then Some l
        else let (a', r') = del_min r in Some ⟨l, a', r'⟩
 | GT ⇒ up_d y l True (del x r))

del_min ⟨l, x, r⟩ =
(if l = ⟨⟩ then (x, r) else let (y, l') = del_min l in (y, ⟨l', x, r⟩))
```

$$\begin{aligned}&up_d\ x\ sib\ twist\ u = (\mathsf{case}\ u\ \mathsf{of}\\ &\qquad\qquad\qquad\qquad None \Rightarrow None\\ &\qquad\qquad\qquad\quad \mid Some\ t \Rightarrow Some\ (node\ twist\ t\ x\ sib))\end{aligned}$$

The top-level deletion function rebalances the root if necessary and maintains the deletion counter:

$$\begin{aligned}&delete\ x\ (t,\ dl) =\\ &(\mathsf{case}\ del\ x\ t\ \mathsf{of}\\ &\quad None \Rightarrow (t,\ dl)\\ &\mid Some\ t' \Rightarrow \mathsf{if}\ bal_d\ |t'|\ (dl+1)\ \mathsf{then}\ (t',\ dl+1)\ \mathsf{else}\ (balance\_tree\ t',\ 0))\end{aligned}$$

6.2 An Efficient Implementation

Just like before, we optimize insertion in two steps. First we cache the size n. That is, the data structure is now a triple $(t,\ n,\ dl)$. Thus $insert_d$ becomes

$$\begin{aligned}&insert_{d2}\ x\ (t,\ n,\ dl) = (\mathsf{case}\ ins_2\ (n+dl)\ 0\ x\ t\ \mathsf{of}\\ &\qquad\qquad\qquad\qquad\quad Same_2 \Rightarrow (t,\ n,\ dl)\\ &\qquad\qquad\qquad\qquad \mid Bal_2\ t' \Rightarrow (t',\ n+1,\ dl))\end{aligned}$$

In another optimization step we call ins_3 instead of ins_2:

$$\begin{aligned}&insert_{d3}\ x\ (t,\ n,\ dl) = (\mathsf{case}\ ins_3\ (n+dl)\ 0\ x\ t\ \mathsf{of}\\ &\qquad\qquad\qquad\qquad\quad Same_2 \Rightarrow (t,\ n,\ dl)\\ &\qquad\qquad\qquad\qquad \mid Bal_2\ t' \Rightarrow (t',\ n+1,\ dl))\end{aligned}$$

Function *delete* is optimized in one step:

$$\begin{aligned}&delete_2\ x\ (t,\ n,\ dl) =\\ &(\mathsf{case}\ del\ x\ t\ \mathsf{of}\\ &\quad None \Rightarrow (t,\ n,\ dl)\\ &\mid Some\ t' \Rightarrow \mathsf{let}\ n' = n-1;\\ &\qquad\qquad\qquad\quad dl' = dl+1\\ &\qquad\qquad\quad \mathsf{in\ if}\ bal_d\ n'\ dl'\ \mathsf{then}\ (t',\ n',\ dl')\ \mathsf{else}\ (bal\_tree\ n'\ t',\ n',\ 0))\end{aligned}$$

Functions $insert_{d3}$ and $delete_2$ are the final top level functions.

6.3 Amortized Complexity

The new potential function Φ_d is the sum of of the previous potential function and an additive term that charges each deletion its share of the overall cost of rebalancing at the root:

$$\Phi_d\ (t,\ n,\ dl) = \Phi\ t + 4 * dl/c_d$$

The factor 4 is due to the cost of *bal_tree* (see Lemma 2).

The amortized complexity of insertion is the same as before:

Theorem 2 $insert_d\ a\ (t,\ dl) = (t',\ dl') \wedge bal_i\ (|t| + dl)\ (height\ t) \longrightarrow$
$t\_insert_{d3}\ a\ (t,\ |t|,\ dl) + \Phi\ t' - \Phi\ t \leq (6 * e + 2) * (height\ t + 1) + 1$

Deletion is similar but its complexity also depends on c_d:

Theorem 3 $t\_delete_2\ x\ (t,\ |t|,\ dl) + \Phi_d\ (delete_2\ x\ (t,\ |t|,\ dl)) - \Phi_d\ (t,\ |t|,\ dl)$
$\leq (6 * e + 1) * height\ t + 4/c_d + 4$

Instantiating the Framework Like in Section 5.3 we instantiate the generic amortized complexity framework: The state space consists of triples $(t,\ n,\ dl)$. The initial state is $(\langle\rangle,\ 0,\ 0)$. The invariant is

$$\lambda(t,\ n,\ dl).\ n = |t| \wedge bal_i\ (|t| + dl)\ (height\ t) \wedge bal_d\ |t|\ dl$$

The potential function is Φ_d. The amortized complexity of $insert_{d3}\ x\ (t,\ n,\ dl)$ is bounded from above by $(6 * e + 2) * (height\ t + 1) + 1$ (Theorem 2). The amortized complexity of $delete_2\ x\ (t,\ n,\ dl)$ is bounded from above by $(6 * e + 1) * height\ t + 4/c_d + 4$ (Theorem 3).

Logarithmic Height We interpret bal_i and e as in definitions (1) and (2) above. However, the proof of logarithmic height in that section no longer works because the invariant $bal_i\ |t|\ (height\ t)$ has become $bal_i\ (|t| + dl)\ (height\ t)$. Following Andersson [2] we introduce another parameter $b > 0$, define

$$c_d = 2^{b/c} - 1 \tag{3}$$

and prove that the invariant implies $height\ t \leq \lceil b + \log_2\ |t|_1 \rceil$. Overall, the amortized complexity is bounded by $(6 * e + 2) * (\lceil b + \log_2\ |t|_1 \rceil + 1) + 1$ (for insertion) and $(6 * e + 1) * \lceil b + \log_2\ |t|_1 \rceil + 4/c_d + 4$ (for deletion).

7 Avoiding Logarithms

The one remaining trouble spot is the logarithm in the computation of bal_i. Implementing it in floating point invalidates the complexity analysis because of rounding errors. As a result, trees may get rebalanced earlier or later than in the mathematical model, which could lead to a different complexity. This was not discussed by any of the previous analyses of this data structure.

We implement $bal_i\ n\ h$ by a table lookup. The idea is to construct a table $bal\_tab :: nat\ list$ such that $bal\_tab\ !\ h$ (where $xs\ !\ n$ is the n-th element of xs, starting with 0) is the least n such that $h \leq \lceil c * \log_2\ (n + 1) \rceil$ and thus

$$bal_i\ n\ h = (bal\_tab\ !\ h \leq n)$$

That is, we have reduced a test involving the log function to a table lookup.

Of course tables are finite. Hence we can only guarantee partial correctness, up to some fixed value of h. But because h is logarithmic in the size of the tree, a small table suffices to hold enough values to cater for any tree that can be stored (for example) in the 64-bit address space.

The definition of bal_tab is straightforward: for a given c, set

$$bal\_tab\ !\ h = \lfloor 2^{(h\ -\ 1)/c} \rfloor \tag{4}$$

for $h = 0$ up to some maximum value. The difficulty is obtaining a verified table because the exponent $(h - 1)/c$ is in general not an integer. We solve this

difficulty by result checking: we compute *bal_tab* externally (e.g. in some programming language), define *bal_tab* with the values obtained, and have Isabelle prove automatically that *bal_tab* is correct. We go through this process step by step.

First we fix some concrete c, compute *bal_tab* up to a sufficiently large size of the tree, and define *bal_tab* in Isabelle. For example, for $c = 3/2$ we have a table of 50 elements if we stop at 2^{33}:

bal_tab = [0, 1, 1, 2, 4, 6, 10, 16, ..., 2705659852, 4294967296]

Then we verify the correctness of *bal_tab* in two steps. First we prove automatically that the values satisfy (4). This relies on a specialized proof method named *approximation* [15] based on interval arithmetic. It can prove propositions like $5 \leq x \leq 7 \longrightarrow \log_2\ x \leq x - 21/10$, in particular if there are no free variables, e.g. $\log_2\ 5 \leq 3/2 * \log_2\ 3$. It proves in a few seconds that

Lemma 8 $\forall\, i{<}\mathit{length}\ \mathit{bal\_tab}.\ \mathit{bal\_tab}\ !\ i = \lfloor 2^{(i\ -\ 1)/c} \rfloor$

By composition with some pre-proved generic lemmas the desired correctness proposition for our concrete *bal_tab* follows:

$h < \mathit{length}\ \mathit{bal\_tab} \longrightarrow \mathit{bal}_i\ n\ h = (\mathit{bal\_tab}\ !\ h \leq n)$

Finally note that although *bal_tab* is a list, it can be implemented as an immutable array.

8 Experimental Results

We have implemented root-balanced trees in Standard ML (with the help of Isabelle's code generator [5,12]) and compared their performance with those of two implementations of AVL and red-black trees [21]. To avoid floating point arithmetic, the tabulation approach from Section 7 was used. Keys are unbounded integers. The code was compiled with Poly/ML 5.6 [17] and executed under Linux on an Intel Core i7-2700K, 3.5 GHz fixed, and 16 GB RAM. We measured the total CPU time used, including garbage collection.

The table in Figure 1 summarizes the results of our measurements. Each of the tests is executed with 10^5 elements in the tree. Each such test case was executed 100 times with each implementation to reduce statistical variations due to randomization and garbage collection.

Each test is executed with two versions of root-balanced trees: one where $c = c_d = 1.2$ and one where $c = c_d = 1.5$. They are called Root-Bal. 1.2 and Root-Bal. 1.5. In principle c and c_d are independent but we have identified them to reduce the number of versions to consider. The identification makes sense because in both cases a larger constant means lazier rebalancing.

We compare two kinds of workloads: uniformly distributed inputs ("Random") and decreasing inputs $n, \ldots, 1$ ("Sorted").

First we look at the upper part of the table with relative timing figures for insertion, deletion and search. The numbers are relative to Root-Bal. 1.5. For

	AVL	Red-Black	Root-Bal. 1.2	Root-Bal. 1.5
Insert Random	1.1	1.7	1.1	1
Insert Sorted	0.3	0.7	1.6	1
Delete Random	1.3	1.6	1.0	1
Delete Sorted	2.0	0.9	1.1	1
Search Random	0.9	1.1	1.0	1
Search Sorted	1.5	1.3	1.0	1
Path Length Random	0.8	0.9	0.9	1
Path Length Sorted	1.0	1.0	1.0	1

Fig. 1. Experimental Results

example, Insert Random with red-black trees takes 1.7 times longer than with root-balanced trees where $c = c_d = 1.5$.

The insertion tests measure how long it takes to insert n elements into an initially empty tree, in random or in sorted order. In Insert Random, the two root-balanced trees beat AVL and red-black trees because root-balanced trees require less restructuring: randomly generated trees are already reasonably balanced. For sorted inputs, the trees get out of balance all the time and partial rebalancing becomes more costly than the local modifications in AVL and red-black trees. This is the only place where our choice of c's has a significant impact.

Deletion starts with the tree created by the corresponding insertion run and deletes all elements in random or sorted order. For random inputs, the performance of all four trees is almost the same. For sorted input, root-balanced trees beat AVL trees and are only slightly slower than red-black trees.

Searches start with the tree created by the corresponding insertion run; all elements in the tree are searched in random or sorted order. With random input there is very little difference between the four search trees. Search Sorted shows a noticeable slowdown for red-black and AVL trees. This could be a cache phenomenon because the nodes of red-black and AVL trees are larger.

Our measurements for insertion and deletion (roughly) confirm those by Galperin and Rivest [11], although they use weight-balanced rather than our height-balanced trees. The situation w.r.t. searches is more complicated. Their Figure 4 shows that with random input, searching in root-balanced trees is 4 times faster than in red-black trees; they do not offer an explanation. In contrast, our data shows very little difference between the different kinds of search trees. Since all the trees we tested are binary search trees, the search time should only depend on their shape (ignoring cache issues, which favour the smaller root-balanced trees). If we search for all the elements in a tree (as we do in our Search tests), the search time should be proportional to the internal path length of the tree (the sum of the lengths of all paths from the root to a node). The last two lines in Figure 1 show that the internal path lengths are relatively close together for Random and practically identical for Sorted. This confirms our measurements of search times and suggests that the discrepancy between root-balanced and other trees in Search Sorted is indeed due to cache behaviour.

9 Related Work

There is a rich literature on resource analysis and we can only mention the most relevant work. The problem of inferring cost functions for functional programs has been studied, for example, by Sands [28] and Vasconcelos and Hammond [30]. Early work on automatic complexity analysis includes Wegbreit's METRIC system [31] for LISP, Le Métayer's ACE system [16] for FP, and Benzinger's ACA [4] system for NUPRL. The recent work by Hoffmann et al. (e.g. [13,14]) is particularly impressive (although currently restricted to polynomials). Type systems are a popular framework for tracking resources [7,8,18]. The last two references follow the same monadic, dependently typed approach in different theorem provers. Our approach is similar but the running times are not tracked on the level of types but on the level of values. However, none of these papers makes an explicit connection to some cost model also formalized in the theorem prover. This is what sets Atkey's work [3] apart. 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. Verified cost analyses for functional language have also been studied [10,9]. 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 references above, only McCarthy *et al.* [18] has examples involving logarithms (instead of merely polynomials) and they are much simpler than root-balanced trees. Like this paper and our earlier work [21], the paper by Charguéraud and Pottier [6] is at the complex, interactive end: they verify the almost-linear amortized complexity of a Union-Find implementation in OCaml in Coq using a separation logic with time credits.

The idea of rebuilding whole substructures of a data structure, but only at intervals, goes back at least to Overmars and van Leeuwen [26,27] who called it *partial rebuilding* and applied it to weight-balanced trees. Partial rebuilding was again applied to weight-balanced trees by Galperin and Rivest [11] where the resulting data structure is called a *scapegoat tree*. We build on Andersson's work [1,2] who realized that one can apply partial rebuilding to trees balanced only at the root, which he called *trees of balanced height* [1] and later *general balanced trees* [2]. We call them *root-balanced* to emphasize the restriction of the balance criterion to the root. All these publications are high-level in that algorithms are described in words and the proofs are based on intuition rather than code. Our proofs roughly follow Andersson [2, Section 3] whose arguments are very high-level. In particular, he does not spell out the potential function. In Section 4 he performs a more precise analysis to obtain smaller constants, but now employing a potential function that can look into the future, an *ad hoc* concept. He argues informally that this concept is appropriate for the problem at hand. In contrast, we provide an explicit potential function of the standard kind. Andersson does not discuss the complications entailed by the logarithm in the balance test.

10 Conclusion

We have presented and verified a functional implementation of the general balanced trees by Andersson [2] in Isabelle. With the help of a lightweight monadic framework for modelling execution time we verified that inserion and deletion have amortized logarithmic complexity. We have also shown how to avoid computing with logarithms, which a direct implementation of Andersson's balance criterion would require. Finally we have presented experimental results showing that root-balanced trees are competitive with AVL and red-black trees.

Acknowledgement Johannes Hölzl suggested the tabulation approach. Manuel Eberl ran the measurements. One of the referees suggested valuable improvements to the time monad.

References

1. Andersson, A.: Improving partial rebuilding by using simple balance criteria. In: Dehne, F., Sack, J.R., Santoro, N. (eds.) Algorithms and Data Structures (WADS '89). LNCS, vol. 382, pp. 393–402. Springer (1989)
2. Andersson, A.: General balanced trees. J. Algorithms 30(1), 1–18 (1999)
3. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods in Computer Science 7(2) (2011)
4. Benzinger, R.: Automated higher-order complexity analysis. Theor. Comput. Sci. 318(1-2), 79–103 (2004)
5. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) Types for Proofs and Programs (TYPES 2000). LNCS, vol. 2277, pp. 24–40. Springer (2002)
6. Charguéraud, A., Pottier, F.: Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 137–153. Springer (2015)
7. Crary, K., Weirich, S.: Resource bound certification. In: Proc. 27th Symposium on Principles of Programming Languages. pp. 184–198. POPL '00, ACM (2000)
8. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Proc. 35th Symposium on Principles of Programming Languages. pp. 133–144. POPL '08, ACM (2008)
9. Danner, N., Licata, D.R., Ramyaa, R.: Denotational cost semantics for functional languages with inductive types. In: Proc. International Conference on Functional Programming. pp. 140–151. ICFP 2015, ACM (2015)
10. Danner, N., Paykin, J., Royer, J.: A static cost analysis for a higher-order language. In: Proc. Workshop Programming Languages Meets Program Verification. pp. 25–34. PLPV '13, ACM (2013)
11. Galperin, I., Rivest, R.L.: Scapegoat trees. In: Ramachandran, V. (ed.) Proc. Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms. pp. 165–174 (1993)
12. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming (FLOPS 2010). LNCS, vol. 6009, pp. 103–117. Springer (2010)

13. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. 34(3), 14 (2012)
14. Hoffmann, J., Das, A., , Weng, S.C.: Towards automatic resource bound analysis for OCaml. In: Proc. 44th Symposium on Principles of Programming Languages. pp. 359–373. POPL '17, ACM (2017)
15. Hölzl, J.: Proving inequalities over reals with computation in Isabelle/HOL. In: Reis, G., Théry, L. (eds.) Programming Languages for Mechanized Mathematics Systems (ACM SIGSAM PLMMS 2009). pp. 38–45 (2009)
16. Le Métayer, D.: ACE: An automatic complexity evaluator. ACM Trans. Program. Lang. Syst. 10(2), 248–266 (1988)
17. Matthews, D.: Poly/ML home page (2017), http://www.polyml.org/
18. McCarthy, J.A., Fetscher, B., New, M.S., Feltey, D., Findler, R.B.: A Coq library for internal verification of running-times. In: Kiselyov, O., King, A. (eds.) Functional and Logic Programming (FLOPS 2016). LNCS, vol. 9613, pp. 144–162. Springer (2016)
19. Nipkow, T.: Order-sorted polymorphism in Isabelle. In: Huet, G., Plotkin, G. (eds.) Logical Environments. pp. 164–188 (1993)
20. Nipkow, T.: Amortized complexity verified. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 310–324. Springer (2015)
21. Nipkow, T.: Automatic functional correctness proofs for functional search trees. In: Blanchette, J., Merz, S. (eds.) Interactive Theorem Proving (ITP 2016). LNCS, vol. 9807, pp. 307–322. Springer (2016)
22. Nipkow, T.: Root-balanced tree. Archive of Formal Proofs (2017), http://isa-afp.org/entries/Root_Balanced_Tree.html, Formal proof development
23. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer (2014), http://concrete-semantics.org
24. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
25. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
26. Overmars, M.: The Design of Dynamic Data Structures, LNCS, vol. 156. Springer (1983)
27. Overmars, M., van Leeuwen, J.: Dynamic multi-dimensional data structures based on quad- and k-d trees. Acta Informatica 17, 267–285 (1982)
28. Sands, D.: Complexity analysis for a lazy higher-order language. In: Jones, N. (ed.) European Symposium on Programming (ESOP). LNCS, vol. 432, pp. 361–376. Springer (1990)
29. Stout, Q.F., Warren, B.L.: Tree rebalancing in optimal time and space. Commun. ACM 29(9), 902–908 (1986)
30. Vasconcelos, P.B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs. In: Trinder, P., Michaelson, G., Pena, R. (eds.) Implementation of Functional Languages, IFL 2003. LNCS, vol. 3145, pp. 86–101. Springer (2004)
31. Wegbreit, B.: Mechanical program analysis. Commun. ACM 18(9), 528–539 (1975)
32. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E., Felty, A. (eds.) Theorem Proving in Higher Order Logics. LNCS, vol. 1275, pp. 307–322. Springer (1997)