Proof Pearl: Braun Trees

Tobias Nipkow[*]
Technical University Munich
Germany
http://www.in.tum.de/~nipkow

Thomas Sewell[†]
Chalmers University of Technology
Sweden
sewell@chalmers.se

Abstract

Braun trees are functional data structures for implementing extensible arrays and priority queues (and sorting functions based on the latter) efficiently. Some well-known functions on Braun trees have not yet been verified, including especially Okasaki's linear time conversion from lists to Braun trees. We supply the missing proofs and verify all of these algorithms in Isabelle, including non-obvious time complexity claims. In particular we provide the first linear-time conversion from Braun trees to lists. We also state and verify a new characterization of Braun trees as the trees t whose index set is the interval $\{1, \ldots, \text{size of } t\}$.

CCS Concepts • **Software and its engineering** → **Software verification**; *Formal software verification*; Functional languages; • **Theory of computation** → *Interactive proof systems*; *Sorting and searching*.

Keywords Braun tree, verification, Isabelle

ACM Reference Format:
Tobias Nipkow and Thomas Sewell. 2020. Proof Pearl: Braun Trees. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372885.3373834

1 Introduction

Braun trees are a popular data structure for implementing extensible arrays in a purely functional manner; they are balanced and thus have optimal logarithmic height. By arrays we mean mappings from an interval of natural numbers and extensible means that we can add new elements at either end. Searching a number in a Braun tree starts at the root and uses the binary representation of the number as a directory string: 0 means "left", 1 means "right".

Braun trees for extensible arrays were first investigated by Braun and Rem [17] and, in a more functional setting, by Hoogerwoord [8]. Okasaki [15] introduced some clever and efficient algorithms for Braun trees. Paulson [16] (in collaboration with Okasaki) presented a different application of Braun trees to implement priority queues. Filliâtre [5] presents the only formal verification we are aware of, verifying a priority queue implementation and one of Okasaki's efficient algorithms using the Why3 system.

In this paper we aim to address the topic of Braun trees comprehensively. We implement all of the operations of interest in Isabelle/HOL. We prove functional correctness of all operations and also verify the more interesting time complexity claims. The Isabelle/HOL sources with the formal proofs can be found partly in the Isabelle distribution in the directory `src/HOL/Data_Structures` and partly in the Archive of Formal Proofs [10].

We make the following contributions:

- The first formal verification of Braun trees (based on Paulson's [16] code) against a specification of arrays.
- The first correctness proofs (formal or informal) of Okasaki's [15] linear time conversion of lists to Braun trees. We also show how to convert a Braun tree into a list in linear time (Oksaki had not covered this direction), which yields an efficient fold function on Braun trees. We formally prove the linear time complexity of the conversions in both directions.
- A novel combinatorial analysis of Braun trees. We show that a tree t is a Braun tree iff the indices of its nodes form the interval $\{1, \ldots, \text{size of } t\}$.
- Proofs of correctness of Okasaki/Paulson [16] priority queues based on Braun trees and of sorting functions built on them, including some proofs of time complexity.

1.1 Isabelle/HOL and Notation

We use the Isabelle/HOL interactive proof assistant [13, 14], including many of its types. Basic types include *bool*, *nat*, *int* and *real*; the function arrow syntax is ⇒. Function lg is the binary logarithm. There are three numeric conversion functions $int :: nat \Rightarrow int$, $nat :: int \Rightarrow nat$ and $real :: nat \Rightarrow real$. We suppress them in this text except where that would result in ambiguities for the reader. The floor and

[*] Supported by DFG Koselleck grant NI 491/16-1
[†] Supported by the Swedish Foundation for Strategic Research.

CPP '20, January 20–21, 2020, New Orleans, LA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7097-4/20/01…$15.00
https://doi.org/10.1145/3372885.3373834

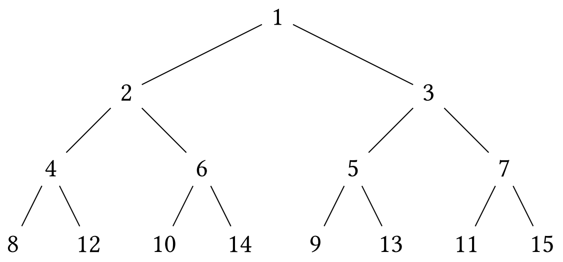


Figure 1. Braun tree with nodes indexed by 1–15.

ceiling conversions $\lfloor x \rfloor$ and $\lceil y \rceil$ convert from *real* to *int* by rounding down and up respectively.

Lists are constructed from the empty list [] via the infix cons-operator (#); the infix (@) appends two lists; $|xs|$ is the length of *xs*; functions *hd* and *tl* return head and tail.

We define binary trees as a recursive data type *'a tree* which has 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 $|t|$ of a tree t is the number of its nodes.

1.2 Inductive Proofs

This paper mostly consists of proofs about recursive functional programs, which are typically shown by induction. To avoid much repetition, when we say we prove a property without giving details, we mean that we proved the result by induction with the help of standard mostly-automated Isabelle proof steps. The value of our presentation lies in providing all the inductive lemmas that are the key challenge when constructing proofs.

2 Braun Trees

Braun trees are binary lookup trees with natural numbers as indices of the nodes. The Braun tree with nodes indexed by 1–15 is shown in Figure 1. The numbers are the indices and not the elements stored in the nodes. Any subset $\{1, \ldots, n\}$ of the nodes, e.g. $\{1, \ldots, 9\}$, also forms a Braun tree. The bits of the binary encoding of the indices tell us how to walk the tree from the root to the corresponding node, starting with the least significant bit. For example, the index 14 is 1110 in binary. If we read it in reverse (least significant) order as left-right-right-stop, we get the path to node 14 in Figure 1.

We do not define a separate type of Braun trees. Instead, we use the general binary tree type *'a tree* mentioned in Section 1.1 and require the following recursive size property:

$braun :: {'a\ tree} \Rightarrow bool$
$braun\ \langle\rangle = True$
$braun\ \langle l, \_, r\rangle$
$= ((|l| = |r| \vee |l| = |r| + 1) \wedge braun\ l \wedge braun\ r)$

The disjunction can alternatively be expressed as $|r| \le |l| \le |r| + 1$. We will call a tree a **Braun tree** iff it satisfies predicate *braun*. We will see (in Section 6) that the predicate is satisfied exactly by those trees t whose nodes are indexed by 1, ..., $|t|$.

The shape of a Braun tree is uniquely determined by its size. This can be expressed by considering trees of type *unit tree* because type *unit* contains exactly one element, (), and thus every node contains the same element. Formally:

Lemma 2.1. *Let* t_1, t_2 :: *unit tree.*
$braun\ t_1 \wedge braun\ t_2 \wedge |t_1| = |t_2| \longrightarrow t_1 = t_2$

2.1 Balance

Braun trees are very precisely balanced: $|r| \le |l| \le |r| + 1$ must hold for every node $\langle l, x, r\rangle$. A more general notion of *balanced* binary trees compares the height of a tree ($h\ t$) with its minimum height ($mh\ t$):

$h :: {'a\ tree} \Rightarrow nat$
$h\ \langle\rangle = 0$
$h\ \langle l, \_, r\rangle = max\ (h\ l)\ (h\ r) + 1$

$mh :: {'a\ tree} \Rightarrow nat$
$mh\ \langle\rangle = 0$
$mh\ \langle l, \_, r\rangle = min\ (mh\ l)\ (mh\ r) + 1$

$balanced :: {'a\ tree} \Rightarrow bool$
$balanced\ t = (h\ t - mh\ t \le 1)$

Isabelle's library includes some basic facts about balanced trees. One is that balanced trees have optimal logarithmic (minimal) height:

$$balanced\ t \longrightarrow h\ t = \lceil \lg\ (|t| + 1) \rceil \tag{1}$$

$$balanced\ t \longrightarrow mh\ t = \lfloor \lg\ (|t| + 1) \rfloor \tag{2}$$

From these two properties we derive a lemma that is rather specifically aimed at Braun trees:

Lemma 2.2. $balanced\ l \wedge balanced\ r \wedge |int\ |l| - int\ |r|| \le 1 \longrightarrow balanced\ \langle l, x, r\rangle$

Proof. The proof is by cases. We consider only the case $|l| = |r| + 1$. From (1) and (2) it follows that $h\ \langle l, x, r\rangle = \lceil \lg\ (|r| + 2) \rceil + 1$ and $mh\ \langle l, x, r\rangle = \lfloor \lg\ (|r| + 1) \rfloor + 1$. These two quantities can be proved to be 1 apart and thus $balanced\ \langle l, x, r\rangle$ holds. □

Now we can prove that all Braun trees are balanced:

Lemma 2.3. $braun\ t \longrightarrow balanced\ t$

The proof is by induction on t and follows directly from Lemma 2.2.

Thus Braun trees have optimal logarithmic height, a fact we will use in the running time analyses that follow.

Another useful property of balanced trees (which follows directly from (1) by monotonicity) is that their height increases monotonically with their size:

Lemma 2.4. $balanced\ t \wedge balanced\ t' \wedge |t| \le |t'| \longrightarrow h\ t \le h\ t'$

Filliâtre [5] proves a variant of this lemma (where *balanced* is replaced by *braun*) directly, without recourse to *balanced*.

With more effort one can prove a stronger version of Lemma 2.4 that essentially says that the height of balanced trees is optimal:

$$balanced\ t \wedge |t| \le |t'| \longrightarrow h\ t \le h\ t'$$

Finally we proved that the height of Braun trees satisfies a similar invariant as the size. As a direct consequence of Lemmas 2.3 and 2.4 we obtain:

Lemma 2.5. $braun\ \langle l, x, r\rangle \longrightarrow h\ r \le h\ l$

Lemma 2.3 together with the trivial $mh\ r \le h\ r$ yields:

Lemma 2.6. $braun\ \langle l, x, r\rangle \longrightarrow h\ l \le h\ r + 1$

3 Arrays

3.1 ADT Specification

Braun trees are useful for encoding arrays indexed by natural numbers. We describe below the expected interface of arrays as an abstract data type (ADT). Arrays can naturally be specified using lists. The Isabelle type *'a list* comes with two array-like operations:

Indexing: $xs\ !\ n$ is the nth element of the list xs.

Updating: $xs[n := x]$ is xs with the nth element replaced by x.

By convention, indexing starts with $n = 0$. If $|xs| \le n$ then $xs\ !\ n$ and $xs[n := x]$ are *underdefined*: they are defined but we do not know what their value is.

We specify the ADT following the model-oriented style of specifications [9]. The ADT includes a collection of operations, an abstraction function, and a representation invariant. The array operations are:

$$\begin{array}{ll} lookup :: {'ar} \Rightarrow nat \Rightarrow {'a} & update :: nat \Rightarrow {'a} \Rightarrow {'ar} \Rightarrow {'ar} \\ len :: {'ar} \Rightarrow nat & array :: {'a}\ list \Rightarrow {'ar} \end{array}$$

The type *'ar* above is the type of the array, and *'a* the type of array elements. The additional invariant $I :: {'ar} \Rightarrow bool$ and abstraction function $list :: {'ar} \Rightarrow {'a}\ list$ are used in the ADT specification. The specification requires that each operation preserves the invariant and behaves like its abstract counterpart on lists:

$$\begin{array}{ll} I\ ar \wedge n < len\ ar \longrightarrow lookup\ ar\ n = list\ ar\ !\ n & (lookup) \\ I\ ar \wedge n < len\ ar \longrightarrow I\ (update\ n\ x\ ar) & (update\text{-}inv) \\ I\ ar \wedge n < len\ ar & \\ \longrightarrow list\ (update\ n\ x\ ar) = (list\ ar)[n := x] & (update) \\ I\ ar \longrightarrow len\ ar = |list\ ar| & (len) \\ I\ (array\ xs) & (array\text{-}inv) \\ list\ (array\ xs) = xs & (array) \end{array}$$

We could have included *list* in the interface as well: it is a useful operation and we will develop an efficient implementation for it.

In Isabelle this ADT is expressed as a *locale* [1]. The ADT can be used in other programs, which are implemented and verified against this abstract interface. The locale mechanism can then instantiate those other programs to a specific instance such as Braun trees.

$$\begin{array}{lcl} lookup\ (t, l)\ n & = & lookup1\ t\ (n + 1) \\ update\ n\ x\ (t, l) & = & (update1\ (n + 1)\ x\ t, l) \\ len\ (t, l) & = & l \\ array\ xs & = & (adds\ xs\ 0\ \langle\rangle, |xs|) \end{array}$$

Figure 2. Array implementation via Braun trees.

3.2 Implementing Arrays via Braun Trees

We start by defining array-like functions on Braun trees. Function $lookup1 :: {'a}\ tree \Rightarrow nat \Rightarrow {'a}$ examines the bits of the index starting from the least significant one:

$$\begin{array}{l} lookup1\ \langle l, x, r\rangle\ n \\ = (\text{if } n = 1 \text{ then } x \\ \quad \text{else } lookup1\ (\text{if } even\ n \text{ then } l \text{ else } r)\ (n\ div\ 2)) \end{array}$$

The least significant bit is the parity of the index and we advance to the next bit by *div* 2. The function is called *lookup*1 rather than *lookup* to emphasize that it expects the index to be at least 1, which simplifies the implementation.

Function $update1 :: nat \Rightarrow {'a} \Rightarrow {'a}\ tree \Rightarrow {'a}\ tree$ descends in the same manner:

$$\begin{array}{l} update1\ n\ x\ \langle\rangle = \langle\langle\rangle, x, \langle\rangle\rangle \\ update1\ n\ x\ \langle l, a, r\rangle \\ = (\text{if } n = 1 \text{ then } \langle l, x, r\rangle \\ \quad \text{else if } even\ n \text{ then } \langle update1\ (n\ div\ 2)\ x\ l, a, r\rangle \\ \qquad \text{else } \langle l, a, update1\ (n\ div\ 2)\ x\ r\rangle) \end{array}$$

The second equation performs the update of existing entries. The first equation, however, creates a new entry and thus supports extending the tree. That is, $update1\ (|t| + 1)\ x\ t$ extends the tree with a new node x at index $|t| + 1$. Function *adds* iterates this process (expecting parameter $n = |t|$) and thus adds a whole list of elements:

$$\begin{array}{l} adds :: {'a}\ list \Rightarrow nat \Rightarrow {'a}\ tree \Rightarrow {'a}\ tree \\ adds\ []\ \_\ t = t \\ adds\ (x\ \#\ xs)\ n\ t = adds\ xs\ (n + 1)\ (update1\ (n + 1)\ x\ t) \end{array}$$

The implementation of the abstract array interface is shown in Figure 2. An array is represented as a pair of a Braun tree and its size, and each operation is a thin wrapper around the Braun tree operation.

3.3 Functional Correctness

Our main result is that the Braun implementation of arrays (in Figure 2) is correct. The invariant is obvious:

$$I\ (t, l) = (braun\ t \wedge l = |t|).$$

The abstraction function $list :: {'a}\ tree \Rightarrow {'a}\ list$ could be defined by repeatedly using *lookup*1. Instead we define *list* recursively:

$list\ \langle\rangle = []$
$list\ \langle l, x, r\rangle = x\ \#\ splice\ (list\ l)\ (list\ r)$

This definition is best explained by looking at Figure 1. The subtrees with root 2 and 3 will be mapped to the lists [2, 4, 6, 8, 10, 12, 14] and [3, 5, 7, 9, 11, 13, 15]. The obvious way to combine these two lists into [2, 3, ..., 15] is to splice them:

$splice :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$
$splice\ []\ ys = ys$
$splice\ (x\ \#\ xs)\ ys = x\ \#\ splice\ ys\ xs$

Before we embark on the actual proofs we state a helpful arithmetic truth (where $\{m..n\} = \{k \mid m \le k \wedge k \le n\}$) that is frequently used implicitly below:

$braun\ \langle l, x, r\rangle \wedge n \in \{1..|\langle l, x, r\rangle|\} \wedge 1 < n \longrightarrow$
$(odd\ n \longrightarrow n\ div\ 2 \in \{1..|r|\}) \wedge (even\ n \longrightarrow n\ div\ 2 \in \{1..|l|\})$

We will now verify that the implementation in Figure 2 satisfies the ADT specification given in Section 3.1.

We start with the ADT property (*len*). First we prove this size lemma (by induction):

$|list\ t| = |t|$

The lemma and the ADT invariant establish property (*len*). We will also use $|list\ t| = |t|$ implicitly in many proofs below.

To establish (*lookup*) we first prove a lemma about *splice*:

$n < |xs| + |ys| \wedge |ys| \le |xs| \wedge |xs| \le |ys| + 1 \longrightarrow$
$splice\ xs\ ys\ !\ n = (\text{if } even\ n \text{ then } xs \text{ else } ys)\ !\ (n\ div\ 2)$

From the lemma we prove this proposition, which establishes the correctness property (*lookup*):

$$braun\ t \wedge i < |t| \longrightarrow list\ t\ !\ i = lookup1\ t\ (i + 1) \quad (3)$$

As a corollary to (3) we obtain that function *list* can indeed be expressed via *lookup*1:

$$braun\ t \longrightarrow list\ t = map\ (lookup1\ t)\ [1..|t|] \quad (4)$$

It follows by **list extensionality**: $xs = ys \longleftrightarrow |xs| = |ys| \wedge (\forall i{<}|xs|.\ xs\ !\ i = ys\ !\ i)$

Let us now verify *update* as implemented via *update*1. We prove two preservation properties which prove (*update-inv*):

$braun\ t \wedge n \in \{1..|t|\} \longrightarrow |update1\ n\ x\ t| = |t|$

$braun\ t \wedge n \in \{1..|t|\} \longrightarrow braun\ (update1\ n\ x\ t)$

The following property relates *lookup*1 and *update*1:

$braun\ t \wedge n \in \{1..|t|\} \longrightarrow$
$lookup1\ (update1\ n\ x\ t)\ m$
$= (\text{if } n = m \text{ then } x \text{ else } lookup1\ t\ m)$

The last three properties together with (4) and list extensionality prove the following proposition, which implies (*update*):

$braun\ t \wedge n \in \{1..|t|\} \longrightarrow$
$list\ (update1\ n\ x\ t) = (list\ t)[n - 1 := x]$

Finally we turn to the constructor *array*. It is implemented in terms of *adds* and *update*1. Their correctness is captured by the following properties whose inductive proofs build on each other:

$$braun\ t \longrightarrow |update1\ (|t| + 1)\ x\ t| = |t| + 1 \quad (5)$$

$$braun\ t \longrightarrow braun\ (update1\ (|t| + 1)\ x\ t) \quad (6)$$

$$braun\ t \longrightarrow list\ (update1\ (|t| + 1)\ x\ t) = list\ t\ @\ [x] \quad (7)$$

$$braun\ t \longrightarrow |adds\ xs\ |t|\ t| = |t| + |xs| \wedge braun\ (adds\ xs\ |t|\ t)$$

$$braun\ t \longrightarrow list\ (adds\ xs\ |t|\ t) = list\ t\ @\ xs$$

The last two of the above imply the remaining proof obligations (*array-inv*) and (*array*). The proof of (7) requires the following properties which are proved by simultaneous induction:

$|ys| \le |xs| \longrightarrow splice\ (xs\ @\ [x])\ ys = splice\ xs\ ys\ @\ [x]$
$|xs| \le |ys| + 1 \longrightarrow splice\ xs\ (ys\ @\ [y]) = splice\ xs\ ys\ @\ [y]$

3.4 Running Time Analysis

The running time of *lookup* and *update* is obviously logarithmic because of the guaranteed logarithmic height of Braun trees. We sketch why *list* and *array* both have running time $O(n \cdot \lg n)$. In the next section we present linear time versions of the latter two functions and prove their complexity formally.

Consider calling *list* on a complete tree of height h. We focus on *splice* because it performs almost all the work. At each level k of the tree (starting with 0 for the root), *splice* is called 2^k times with lists of size 2^{h-k-1}. The running time of *splice* with lists of the same length is proportional to the size of the lists. Thus the running time at each level is $O(2^k \cdot 2^{h-k-1}) = O(2^{h-1}) = O(2^h)$. Thus all the splices together require time $O(h \cdot 2^h)$. Because complete trees have size $n = 2^h$, the bound $O(n \cdot \lg n)$ follows.

Function *array* is implemented via *adds* and thus via repeated calls of *update*1. How expensive is it to call *update*1 n times on a growing tree starting with a leaf? Because *update*1 has logarithmic running time, the n calls roughly take time proportional to $\lg 1 + \cdots + \lg n = \lg(n!)$. Stirling's formula tells us that $\lg(n!) \in \Theta(n \cdot \lg n)$.

4 Flexible Arrays

4.1 ADT Specification

Flexible arrays can be grown and shrunk at either end. The new flexible array ADT extends the previous array ADT with four new operations:

$add\_lo :: 'a \Rightarrow 'ar \Rightarrow 'ar \qquad del\_lo :: 'ar \Rightarrow 'ar$
$add\_hi :: 'a \Rightarrow 'ar \Rightarrow 'ar \qquad del\_hi :: 'ar \Rightarrow 'ar$

These operations must also preserve I and match the behaviour of their counterparts on lists. The *tl* and *butlast* operations below remove the first and last elements of a list.

$$\begin{array}{ll}
I\ ar \longrightarrow I\ (\mathit{add\_lo}\ a\ ar) & (\mathit{add\_lo\text{-}inv}) \\
I\ ar \longrightarrow \mathit{list}\ (\mathit{add\_lo}\ a\ ar) = a\ \#\ \mathit{list}\ ar & (\mathit{add\_lo}) \\
I\ ar \longrightarrow I\ (\mathit{del\_lo}\ ar) & (\mathit{del\_lo\text{-}inv}) \\
I\ ar \longrightarrow \mathit{list}\ (\mathit{del\_lo}\ ar) = \mathit{tl}\ (\mathit{list}\ ar) & (\mathit{del\_lo}) \\
I\ ar \longrightarrow I\ (\mathit{add\_hi}\ a\ ar) & (\mathit{add\_hi\text{-}inv}) \\
I\ ar \longrightarrow \mathit{list}\ (\mathit{add\_hi}\ a\ ar) = \mathit{list}\ ar\ @\ [a] & (\mathit{add\_hi}) \\
I\ ar \longrightarrow I\ (\mathit{del\_hi}\ ar) & (\mathit{del\_hi\text{-}inv}) \\
I\ ar \longrightarrow \mathit{list}\ (\mathit{del\_hi}\ ar) = \mathit{butlast}\ (\mathit{list}\ ar) & (\mathit{del\_hi})
\end{array}$$

4.2 Implementation via Braun Trees

We have already seen that *update*1 adds an element at the high end of a Braun tree. The inverse operation *del_hi* removes the high end, assuming that the given index is the size of the tree:

$$\begin{array}{l}
\mathit{del\_hi} :: \mathit{nat} \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \\
\mathit{del\_hi}\ \_\ \langle\rangle = \langle\rangle \\
\mathit{del\_hi}\ n\ \langle l, x, r\rangle \\
= (\text{if } n = 1 \text{ then } \langle\rangle \\
\quad \text{else if } \mathit{even}\ n \text{ then } \langle \mathit{del\_hi}\ (n\ \mathit{div}\ 2)\ l, x, r\rangle \\
\qquad \text{else } \langle l, x, \mathit{del\_hi}\ (n\ \mathit{div}\ 2)\ r\rangle)
\end{array}$$

It is perhaps intuitive how to place a new node at the bottom of the tree but less clear how to extend the array at the low end since the existing entries all move to new positions. However, Braun trees support a logarithmic implementation:

$$\begin{array}{l}
\mathit{add\_lo} :: {'a} \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \\
\mathit{add\_lo}\ x\ \langle\rangle = \langle\langle\rangle, x, \langle\rangle\rangle \\
\mathit{add\_lo}\ x\ \langle l, a, r\rangle = \langle \mathit{add\_lo}\ a\ r, x, l\rangle
\end{array}$$

Function *add_lo* installs the new element x at the root of the tree. Because the indices of the existing elements change by 1, the left subtree (indices 2, 4, ...) and right subtree (indices 3, 5, ...) change places. The old root, now at index 2, is added to the new left subtree.

Function *del_lo* simply reverses *add_lo* by removing the root and merging the subtrees:

$$\begin{array}{l}
\mathit{del\_lo} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \\
\mathit{del\_lo}\ \langle\rangle = \langle\rangle \\
\mathit{del\_lo}\ \langle l, \_, r\rangle = \mathit{merge}\ l\ r
\end{array}$$

$$\begin{array}{l}
\mathit{merge} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \\
\mathit{merge}\ \langle\rangle\ r = r \\
\mathit{merge}\ \langle l, a, r\rangle\ rr = \langle rr, a, \mathit{merge}\ l\ r\rangle
\end{array}$$

Figure 3 shows the obvious implementation of the operations of the flexible array ADT (on the left-hand side) using the corresponding Braun tree operations (on the right-hand side). It is an extension of the basic array implementation from Figure 2. All these functions have logarithmic time complexity because the Braun tree functions each descend along one branch of the tree.

4.3 Functional Correctness

We now have to prove the correctness properties of the flexible array ADT (Section 4.1). We have already dealt with *update*1 and thus *add_hi* above. Properties (*add_hi-inv*) and (*add_hi*) follow from (5), (6) and (7) stated earlier.

$$\begin{array}{lcl}
\mathit{add\_lo}\ x\ (t, l) & = & (\mathit{add\_lo}\ x\ t, l + 1) \\
\mathit{add\_hi}\ x\ (t, l) & = & (\mathit{update}1\ (l + 1)\ x\ t, l + 1) \\
\mathit{del\_lo}\ (t, l) & = & (\mathit{del\_lo}\ t, l - 1) \\
\mathit{del\_hi}\ (t, l) & = & (\mathit{del\_hi}\ l\ t, l - 1)
\end{array}$$

Figure 3. Flexible array implementation via Braun trees.

We establish the correctness of *del_hi* by proving the following two properties:

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{del\_hi}\ |t|\ t)$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{del\_hi}\ |t|\ t) = \mathit{butlast}\ (\mathit{list}\ t) \tag{8}$$

Our proof of (8) starts with two auxiliary lemmas, the simple fact $\mathit{list}\ t = [] \longleftrightarrow t = \langle\rangle$ and also the following property which relates *splice* to *butlast*:

$$\begin{array}{l}
\mathit{butlast}\ (\mathit{splice}\ xs\ ys) \\
= (\text{if } |ys| < |xs| \text{ then } \mathit{splice}\ (\mathit{butlast}\ xs)\ ys \\
\quad \text{else } \mathit{splice}\ xs\ (\mathit{butlast}\ ys))
\end{array}$$

The ADT correctness property (*del_hi*) follows.

Correctness of *add_lo* on Braun trees is captured by the following two properties:

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{add\_lo}\ x\ t)$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{add\_lo}\ a\ t) = a\ \#\ \mathit{list}\ t$$

Properties (*add_lo-inv*) and (*add_lo*) follow directly.

Finally we turn to *del_lo*. Inductions (for *merge*) and case analyses (for *del_lo*) yield the following correctness properties:

$$\mathit{braun}\ \langle l, x, r\rangle \longrightarrow \mathit{braun}\ (\mathit{merge}\ l\ r)$$

$$\mathit{braun}\ \langle l, x, r\rangle \longrightarrow \mathit{list}\ (\mathit{merge}\ l\ r) = \mathit{splice}\ (\mathit{list}\ l)\ (\mathit{list}\ r)$$

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{del\_lo}\ t)$$

$$\mathit{braun}\ t \longrightarrow \mathit{list}\ (\mathit{del\_lo}\ t) = \mathit{tl}\ (\mathit{list}\ t)$$

The last two properties imply (*del_lo-inv*) and (*del_lo*), and conclude our proof that the ADT implementation is correct.

5 Bigger, Better, Faster, More!

This section is inspired by Okasaki’s [15] efficient algorithms on Braun trees. Our emphasis is on the functions converting between Braun trees and lists. We shall see that their correctness proofs are not trivial and rely on a tricky auxiliary notion *braun_list*. Our function for converting a list to a tree is based on the ideas of the corresponding function by Okasaki but the code is quite different. Okasaki provides no efficient function in the other direction but we do.

For completeness reasons we also verified Okasaki’s functions *size* and *copy*2 (*size_fast* and *braun*2*_of* below) although the functions and proofs are quite simple, the proofs are already given (or suggested) by Okasaki, and Filliâtre has verified the *size_fast* proof in Why3 [5].

Okasaki presents the following $O(\log^2(|t|))$ time function to compute the size:

$$\begin{array}{l}
\textit{size\_fast} :: \textit{'a tree} \Rightarrow \textit{nat}\\
\textit{size\_fast}\ \langle\rangle = 0\\
\textit{size\_fast}\ \langle l, \_, r\rangle = (\text{let } n = \textit{size\_fast}\ r \text{ in } 1 + 2 \cdot n + \textit{diff}\ l\ n)
\end{array}$$

$$\begin{array}{l}
\textit{diff} :: \textit{'a tree} \Rightarrow \textit{nat} \Rightarrow \textit{nat}\\
\textit{diff}\ \langle\rangle\ \_ = 0\\
\textit{diff}\ \langle l, \_, r\rangle\ n\\
= (\text{if } n = 0 \text{ then } 1\\
\quad \text{else if } \textit{even}\ n \text{ then } \textit{diff}\ r\ (n\ \textit{div}\ 2 - 1) \text{ else } \textit{diff}\ l\ (n\ \textit{div}\ 2))
\end{array}$$

Correctness ($\textit{braun}\ t \longrightarrow \textit{size\_fast}\ t = |t|$) follows from this auxiliary property of *diff*:

$$\textit{braun}\ t \wedge |t| \in \{n, n + 1\} \longrightarrow \textit{diff}\ t\ n = |t| - n$$

A simple fact not mentioned by Okasaki is that the height of a Braun tree can be computed in logarithmic time:

$$\begin{array}{l}
\textit{lh} :: \textit{'a tree} \Rightarrow \textit{nat}\\
\textit{lh}\ \langle\rangle = 0\\
\textit{lh}\ \langle l, \_, \_\rangle = \textit{lh}\ l + 1
\end{array}$$

The reason is Lemma 2.5. It allows us to prove that on Braun trees, *lh* computes the height:

$$\textit{braun}\ t \longrightarrow \textit{lh}\ t = h\ t$$

5.1 Initializing a Braun Tree with a Fixed Value

We have so far considered the construction of a Braun tree from a list. Alternatively one may want to create a tree (array) where all elements are initialized to the same value. Okasaki presents function *braun2_of* (which he calls *copy2*) that shares trees as much as possible by producing trees of size n and $n + 1$ in parallel:

$$\begin{array}{l}
\textit{braun2\_of} :: \textit{'a} \Rightarrow \textit{nat} \Rightarrow \textit{'a tree} \times \textit{'a tree}\\
\textit{braun2\_of}\ x\ n\\
= (\text{if } n = 0 \text{ then } (\langle\rangle, \langle\langle\rangle, x, \langle\rangle\rangle)\\
\quad \text{else let } (s, t) = \textit{braun2\_of}\ x\ ((n - 1)\ \textit{div}\ 2)\\
\qquad\quad \text{in if } \textit{odd}\ n \text{ then } (\langle s, x, s\rangle, \langle t, x, s\rangle)\\
\qquad\qquad \text{else } (\langle t, x, s\rangle, \langle t, x, t\rangle))
\end{array}$$

$$\begin{array}{l}
\textit{braun\_of} :: \textit{'a} \Rightarrow \textit{nat} \Rightarrow \textit{'a tree}\\
\textit{braun\_of}\ x\ n = \textit{fst}\ (\textit{braun2\_of}\ x\ n)
\end{array}$$

The running time is clearly logarithmic in n.

The correctness properties are:

$$\textit{list}\ (\textit{braun\_of}\ x\ n) = \textit{replicate}\ n\ x \quad \text{and} \quad \textit{braun}\ (\textit{braun\_of}\ x\ n)$$

where $\textit{replicate}\ n\ x$ is a list of n copies of x. These are corollaries of the more general inductive statement:

$$\begin{array}{l}
\textit{braun2\_of}\ x\ n = (s, t) \longrightarrow\\
\textit{list}\ s = \textit{replicate}\ n\ x \wedge \textit{list}\ t = \textit{replicate}\ (n + 1)\ x\\
\wedge\ |s| = n \wedge |t| = n + 1 \wedge \textit{braun}\ s \wedge \textit{braun}\ t
\end{array}$$

5.2 Converting a List into a Braun Tree

We improve on function *adds* from Section 3.2 that has running time $\Theta(n \cdot \lg n)$ by developing a linear-time function. Given a list of elements $[1, 2, \ldots]$, we can subdivide it into sublists $[1], [2, 3], [4, \ldots, 7], \ldots$ such that the kth sublist contains the elements of level k of the corresponding Braun tree. This is simply because on each level we have the entries whose index has $k + 1$ bits. Thus we need to process the input list in chunks of size 2^k to produce the trees on level k. For reasons of space we must refer the reader to Okasaki who presents a good example-based explanation how these chunks need to be processed. We simply present the definition of our main function $\textit{brauns} :: \textit{nat} \Rightarrow \textit{'a list} \Rightarrow \textit{'a tree list}$. Loosely speaking, $\textit{brauns}\ k\ \textit{xs}$ produces the Braun trees on level k.

$$\begin{array}{l}
\textit{brauns}\ k\ \textit{xs}\\
= (\text{if } \textit{xs} = [] \text{ then } []\\
\quad \text{else let } \textit{ys} = \textit{take}\ 2^k\ \textit{xs};\ \textit{zs} = \textit{drop}\ 2^k\ \textit{xs};\\
\qquad\qquad\quad \textit{ts} = \textit{brauns}\ (k + 1)\ \textit{zs}\\
\qquad\quad \text{in } \textit{nodes}\ \textit{ts}\ \textit{ys}\ (\textit{drop}\ 2^k\ \textit{ts}))
\end{array}$$

Function *brauns* chops off a chunk *ys* of size 2^k from the input list, and recursively converts the remainder of the list into a list *ts* of (at most) 2^{k+1} trees. This list is (conceptually) split into $\textit{take}\ 2^k\ \textit{ts}$ and $\textit{drop}\ 2^k\ \textit{ts}$ which are combined with *ys* by function *nodes* that traverses its three argument lists simultaneously. As a local optimization, we pass all of *ts* rather than just $\textit{take}\ 2^k\ \textit{ts}$ to *nodes*.

$$\begin{array}{l}
\textit{nodes} :: \textit{'a tree list} \Rightarrow \textit{'a list} \Rightarrow \textit{'a tree list} \Rightarrow \textit{'a tree list}\\
\textit{nodes}\ (l\ \#\ \textit{ls})\ (x\ \#\ \textit{xs})\ (r\ \#\ \textit{rs}) = \langle l, x, r\rangle\ \#\ \textit{nodes}\ \textit{ls}\ \textit{xs}\ \textit{rs}\\
\textit{nodes}\ (l\ \#\ \textit{ls})\ (x\ \#\ \textit{xs})\ [] = \langle l, x, \langle\rangle\rangle\ \#\ \textit{nodes}\ \textit{ls}\ \textit{xs}\ []\\
\textit{nodes}\ []\ (x\ \#\ \textit{xs})\ (r\ \#\ \textit{rs}) = \langle\langle\rangle, x, r\rangle\ \#\ \textit{nodes}\ []\ \textit{xs}\ \textit{rs}\\
\textit{nodes}\ []\ (x\ \#\ \textit{xs})\ [] = \langle\langle\rangle, x, \langle\rangle\rangle\ \#\ \textit{nodes}\ []\ \textit{xs}\ []\\
\textit{nodes}\ \_\ []\ \_ = []
\end{array}$$

The final row of a Braun tree will usually be incomplete, which results in function *nodes* processing lists of different lengths. It handles these cases by implicitly extending the row with additional $\langle\rangle$ elements as necessary.

The top-level function $\textit{brauns1} :: \textit{'a list} \Rightarrow \textit{'a tree}$ for turning a list into a tree simply extracts the first (and only) element from the list computed by $\textit{brauns}\ 0$:

$$\textit{brauns1}\ \textit{xs} = (\text{if } \textit{xs} = [] \text{ then } \langle\rangle \text{ else } \textit{brauns}\ 0\ \textit{xs}\ !\ 0)$$

5.2.1 Functional Correctness

The key correctness lemma below expresses a property of Braun trees: the subtrees on level k consist of all elements of the input list *xs* that are 2^k elements apart, starting from some offset. The most concise definition is this one

$$\textit{take\_nths}\ i\ k\ \textit{xs} = \textit{nths}\ \textit{xs}\ (\textstyle\bigcup_n \{n \cdot 2^k + i\})$$

where $\textit{nths}\ [x_0,x_1,\ldots]\ I$ is the list of all x_i (in increasing order of indices) such that $i \in I$. However, we found that proof automation increases with this recursive definition which implies the preceding specification in terms of *nths*:

$$\begin{array}{l}
\textit{take\_nths} :: \textit{nat} \Rightarrow \textit{nat} \Rightarrow \textit{'a list} \Rightarrow \textit{'a list}\\
\textit{take\_nths}\ \_\ \_\ [] = []\\
\textit{take\_nths}\ i\ k\ (x\ \#\ \textit{xs})\\
= (\text{if } i = 0 \text{ then } x\ \#\ \textit{take\_nths}\ (2^k - 1)\ k\ \textit{xs}
\end{array}$$

$\quad$ else $take\_nths\ (i - 1)\ k\ xs)$

A number of simple properties follow by easy induction:

$$take\_nths\ i\ k\ (drop\ j\ xs) = take\_nths\ (i + j)\ k\ xs \tag{9}$$

$$take\_nths\ 0\ 0\ xs = xs \tag{10}$$

$$splice\ (take\_nths\ 0\ 1\ xs)\ (take\_nths\ 1\ 1\ xs) = xs \tag{11}$$

$$\begin{gathered} take\_nths\ i\ m\ (take\_nths\ j\ n\ xs) \\ = take\_nths\ (i \cdot 2^n + j)\ (m + n)\ xs \end{gathered} \tag{12}$$

$$take\_nths\ i\ k\ xs = [\,] \longleftrightarrow |xs| \le i \tag{13}$$

$$i < |xs| \longrightarrow hd\ (take\_nths\ i\ k\ xs) = xs\ !\ i \tag{14}$$

$$\begin{gathered} |xs| = |ys| \lor |xs| = |ys| + 1 \longrightarrow \\ take\_nths\ 0\ 1\ (splice\ xs\ ys) = xs\ \land \\ take\_nths\ 1\ 1\ (splice\ xs\ ys) = ys \end{gathered} \tag{15}$$

$$\begin{gathered} |take\_nths\ 0\ 1\ xs| = |take\_nths\ 1\ 1\ xs|\ \lor \\ |take\_nths\ 0\ 1\ xs| = |take\_nths\ 1\ 1\ xs| + 1 \end{gathered} \tag{16}$$

We also introduce a predicate $braun\_list :: {'a}\ tree \Rightarrow {'a}\ list \Rightarrow bool$:

$braun\_list\ \langle\rangle\ xs = (xs = [\,])$
$braun\_list\ \langle l, x, r\rangle\ xs$
$= (xs \ne [\,] \land x = hd\ xs \land braun\_list\ l\ (take\_nths\ 1\ 1\ xs)\ \land$
$braun\_list\ r\ (take\_nths\ 2\ 1\ xs))$

This definition may look a bit mysterious at first. The idea is that instead of relating $\langle l, x, r\rangle$ to xs via $splice$ we invert the process and relate l and r to the even and odd numbered elements of $drop\ 1\ xs$. Luckily $braun\_list$ satisfies a simple specification:

Lemma 5.1. $braun\_list\ t\ xs \longleftrightarrow braun\ t \land xs = list\ t$

Proof. The proof is by induction on t. The base case is trivial. In the induction step we use (16) to prove $braun\ t$ and (11) and (15) to prove $xs = list\ t$. □

The correctness proof of $brauns$ needs these lemmas:

$$|nodes\ ls\ xs\ rs| = |xs| \tag{17}$$

$$\begin{gathered} i < |xs| \longrightarrow \\ nodes\ ls\ xs\ rs\ !\ i \\ = \langle \text{if } i < |ls| \text{ then } ls\ !\ i \text{ else } \langle\rangle,\ xs\ !\ i, \\ \text{if } i < |rs| \text{ then } rs\ !\ i \text{ else } \langle\rangle\rangle \end{gathered} \tag{18}$$

$$|brauns\ k\ xs| = min\ |xs|\ 2^k \tag{19}$$

Lemmas (17) and (18) capture the correctness of $nodes$, returning tree nodes built from the input lists padded with $\langle\rangle$ elements.

The main theorem expresses the following correctness property of the elements of $brauns\ k\ xs$: every tree $brauns\ k\ xs\ !\ i$ is a Braun tree and its list of elements is $take\_nths\ i\ k\ xs$:

Theorem 5.2. $i < min\ |xs|\ 2^k \longrightarrow$
$braun\_list\ (brauns\ k\ xs\ !\ i)\ (take\_nths\ i\ k\ xs)$

Proof. The proof is by induction on the length of xs. Assume $i < min\ |xs|\ 2^k$, which implies $xs \ne [\,]$. Let $zs = drop\ 2^k\ xs$. Thus $|zs| < |xs|$ and therefore the IH applies to zs and yields the property

$$\begin{gathered} \forall\, i\ j.\ j = i + 2^k \land i < min\ |zs|\ 2^{k+1} \longrightarrow \\ braun\_list\ (ts\ !\ i)\ (take\_nths\ j\ (k + 1)\ xs) \end{gathered} \tag{$*$}$$

where $ts = brauns\ (k + 1)\ zs$. Let $ts' = drop\ 2^k\ ts$.

Since $xs \ne [\,]$, $brauns\ k\ xs\ !\ i$ is by definition $nodes\ ts\ (take\ 2^k\ xs)\ ts'\ !\ i$, which we can examine via (18). This results in two conditionals and thus four possible cases, all of which can be solved by rewriting with (*), lemmas (18), (12), (13), (14), (19) and assumptions. □

Setting $i = k = 0$ in this theorem yields the correctness of $brauns1$ using Lemma 5.1 and (10):

Corollary 5.3. $braun\ (brauns1\ xs) \land list\ (brauns1\ xs) = xs$

5.2.2 Running Time Analysis

We will analyse running time by defining for each function f a timing function t_f that takes the same arguments as f but computes the number of function calls the computation of f needs, the 'time'. Function t_f follows the same recursion structure as f and can be seen as an abstract interpretation of f. This is similar to our previous work [11] however for simplicity of presentation we will define each f and t_f directly rather than deriving them from a monadic function that computes both the value and the time. We must convince ourselves that these timing functions are representative of real execution time, which is usually clear.

We focus on the key function $brauns$. In the step from $brauns$ to t_brauns we simplify matters a little bit: we count only the expensive operations that traverse lists and ignore the other small additive constants. The time to evaluate $take\ n\ xs$ and $drop\ n\ xs$ is linear in $min\ n\ |xs|$ and we simply use $min\ n\ |xs|$. Thus the three $take$ and $drop$ calls contribute $3 \cdot min\ 2^k\ |xs|$. Evaluating $nodes\ \_\ ys\ \_$ takes time linear in $|ys| = |take\ 2^k\ xs| = min\ 2^k\ |xs|$. Thus we obtain the following definition:

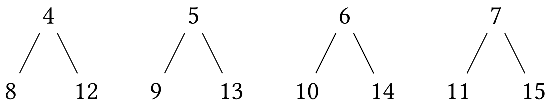
```
$t\_brauns :: nat \Rightarrow {'a}\ list \Rightarrow nat$
$t\_brauns\ k\ xs$
$= ($if $xs = [\,]$ then $0$
   else let $ys = take\ 2^k\ xs$; $zs = drop\ 2^k\ xs$;
            $ts = brauns\ (k + 1)\ zs$
        in $4 \cdot min\ 2^k\ |xs| + t\_brauns\ (k + 1)\ zs)$
```

Lemma 5.4. $t\_brauns\ k\ xs = 4 \cdot |xs|$

Proof. The proof is by induction on the length of xs. If $xs = [\,]$ the claim is trivial. If $xs \ne [\,]$ the claim follows by IH and the fact $|drop\ n\ xs| = |xs| - n$. □

5.3 Converting a Braun Tree into a List

We improve on function $list$ that has running time $O(n \cdot \lg n)$ by developing a linear-time version. Imagine that we want to invert the computation of $brauns1$ and thus of $brauns$. We convert a whole list of trees. Consider the last two levels of the tree in Figure 1 and reorder them by increasing root labels:



The following strategy strongly suggests itself: first the roots, then the left subtrees, then the right subtrees. The recursive application of this strategy also takes care of the required reordering of the subtrees. Of course we have to ignore any leaves we encounter. This is the resulting function:

$list\_fast\_rec :: {'a}\ tree\ list \Rightarrow {'a}\ list$
$list\_fast\_rec\ ts$
$= (\text{let } us = filter\ (\lambda t.\ t \neq \langle\rangle)\ ts$
$\quad \text{in if } us = [\,] \text{ then } [\,]$
$\qquad \text{else } map\ value\ us\ @$
$\qquad\qquad list\_fast\_rec\ (map\ left\ us\ @\ map\ right\ us))$

where $value\ \langle l, x, r\rangle = x$, $left\ \langle l, x, r\rangle = l$ and $right\ \langle l, x, r\rangle = r$.

To prove the termination of *list_fast_rec* we must supply the measure function $\varphi = sum\_list \circ map\ tree\_size$, the sum of the sizes of the trees in the list. The proof also requires an auxiliary lemma, which we skip here.

The top level function $list\_fast :: {'a}\ tree \Rightarrow {'a}\ list$ extracts a list from a single tree:

$list\_fast\ t = list\_fast\_rec\ [t]$

From *list_fast* one can easily derive an efficient fold function on Braun trees that processes the elements in the tree in the order of their indices.

5.3.1 Functional Correctness

We want to prove correctness of *list_fast*: $list\_fast\ t = list\ t$ if $braun\ t$. A direct proof of $list\_fast\_rec\ [t] = list\ t$ will fail and we need to generalize this statement to all lists of trees of length 2^k. Reusing the infrastructure from the previous subsection this can be expressed as follows:

Theorem 5.5.
$|ts| = 2^k \wedge (\forall\, i{<}2^k.\ braun\_list\ (ts\ !\ i)\ (take\_nths\ i\ k\ xs)) \longrightarrow$
$\quad list\_fast\_rec\ ts = xs$

Proof. The proof is by induction on the length of *xs*. Assume the two premises. There are two cases. First assume $|xs| < 2^k$. Then

$$ts = map\ (\lambda x.\ \langle\langle\rangle, x, \langle\rangle\rangle)\ xs\ @\ replicate\ n\ \langle\rangle \qquad (*)$$

where $n = |ts| - |xs|$. This can be proved pointwise. Take some $i < 2^k$. If $i < |xs|$ then $take\_nths\ i\ k\ xs = take\ 1\ (drop\ i\ xs)$ (which can be proved by induction on *xs*). By definition of *braun_list* it follows that $t\ !\ i = \langle l, xs\ !\ i, r\rangle$ for some l and r such that $braun\_list\ l\ [\,]$ and $braun\_list\ r\ [\,]$ and thus $l = r = \langle\rangle$, i.e. $t\ !\ i = \langle\langle\rangle, xs\ !\ i, \langle\rangle\rangle$. If $\neg\ i < |xs|$ then $take\_nths\ i\ k\ xs = [\,]$ by (13) and thus $braun\_list\ (ts\ !\ i)\ [\,]$ by the second premise and thus $ts\ !\ i = \langle\rangle$ by definition of *braun_list*. This concludes the proof of (∗). The desired $list\_fast\_rec\ ts = xs$ follows easily by definition of *list_fast_rec*.

Now assume $\neg\ |xs| < 2^k$. Then for all $i < 2^k$

$ts\ !\ i \neq \langle\rangle \wedge value\ (ts\ !\ i) = xs\ !\ i\ \wedge$
$braun\_list\ (left\ (ts\ !\ i))\ (take\_nths\ (i + 2^k)\ (k + 1)\ xs)\ \wedge$
$braun\_list\ (right\ (ts\ !\ i))\ (take\_nths\ (i + 2 \cdot 2^k)\ (k + 1)\ xs)$

follows from the second premise with the help of (12), (13) and (14). We obtain two consequences:

$map\ root\_val\ ts = take\ 2^k\ xs$
$list\_fast\_rec\ (map\ left\ ts\ @\ map\ right\ ts) = drop\ 2^k\ xs$

The first consequence follows by pointwise reasoning, the second consequence with the help of the IH and (9). From these two consequences the desired conclusion $list\_fast\_rec\ ts = xs$ follows by definition of *list_fast_rec*. □

5.3.2 Running Time Analysis

We focus on *list_fast_rec*. In the step from *list_fast_rec* to *t_list_fast_rec* we simplify matters a little bit: we count only the expensive operations that traverse lists and ignore the other small additive constants. The time to evaluate $map\ left\ ts$, $map\ right\ ts$, $filter\ (\lambda t.\ t \neq \langle\rangle)\ ts$ and $ts\ @\ ts'$ is linear in $|ts|$ and we simply use $|ts|$. As a result we obtain the following definition of *t_list_fast_rec*:

$t\_list\_fast\_rec :: {'a}\ tree\ list \Rightarrow nat$
$t\_list\_fast\_rec\ ts$
$= (\text{let } us = filter\ (\lambda t.\ t \neq \langle\rangle)\ ts$
$\quad \text{in } |ts|\ +$
$\qquad (\text{if } us = [\,] \text{ then } 0$
$\qquad\ \text{else } 5 \cdot |us|\ +$
$\qquad\qquad t\_list\_fast\_rec\ (map\ left\ us\ @\ map\ right\ us)))$

The following inductive property is an abstraction of the core of the termination argument of *list_fast_rec* above.

$(\forall\, t{\in}set\ ts.\ t \neq \langle\rangle) \longrightarrow$
$(\sum t{\leftarrow}ts.\ k \cdot |t|)$
$$= (\textstyle\sum t{\leftarrow}map\ left\ ts\ @\ map\ right\ ts.\ k \cdot |t|) + k \cdot |ts| \qquad (20)$$

The Haskell-inspired notation $\sum x{\leftarrow}xs.\ f\ x$ is syntactic sugar for $sum\_list\ (map\ f\ xs)$.

Now we can state and prove a linear upper bound of *t_list_fast_rec*:

Theorem 5.6. $t\_list\_fast\_rec\ ts \leq (\sum t{\leftarrow}ts.\ 7 \cdot |t| + 1)$

Proof. The proof is by induction on the sum of the sizes of the trees in *ts*, which decreases with recursive calls as we proved above. If $ts = [\,]$ the claim is trivial. Now assume $ts \neq [\,]$ and let $us = filter\ (\lambda t.\ t \neq \langle\rangle)\ ts$ and $children = map\ left\ us\ @\ map\ right\ us$.

$t\_list\_fast\_rec\ ts = t\_list\_fast\_rec\ children + 5 \cdot |us| + |ts|$
$\leq (\sum t{\leftarrow}children.\ 7 \cdot |t| + 1) + 5 \cdot |us| + |ts|$ by IH
$= (\sum t{\leftarrow}children.\ 7 \cdot |t|) + 7 \cdot |us| + |ts|$
$= (\sum t{\leftarrow}us.\ 7 \cdot |t|) + |ts|$ by (20)
$\leq (\sum t{\leftarrow}ts.\ 7 \cdot |t|) + |ts| = (\sum t{\leftarrow}ts.\ 7 \cdot |t| + 1)$ □

5.4 Generalisation to Other Tries

A Braun tree is an instance of a more general structure, a trie [3, 6]. A trie is a search tree where the path followed during a lookup is uniquely determined by the key being looked up. The key ideas of the list conversions *brauns* and *list_fast_rec* can be adapted to some other tries.

One such trie is the "sptree" datatype provided by the standard library of the HOL4 theorem prover [18]. This trie has similar lookup structure to a Braun tree, but it may be sparsely populated and it may be unbalanced. We have adapted the concepts of *list_fast_rec* to the sparse case, converting to a sorted association list (a list of index/element pairs). This adds a previously missing operation to the HOL4 library. It does not seem to be worthwhile to adapt the approach of *brauns* however, because of the time cost of comparing indices while assembling a row.

6 More Combinatorics of Braun Trees

This section gives an alternative characterization of Braun trees that seems to have gone unnoticed in the literature. It is based on the notion of the index set of a tree, defined below. The image of a set S under a function f is defined by $f\,`\,S = \{y \mid \exists\, x{\in}S.\ y = f\,x\}$.

$\mathit{braun\_indices} :: {'a}\ \mathit{tree} \Rightarrow \mathit{nat}\ \mathit{set}$
$\mathit{braun\_indices}\ \langle\rangle = \{\}$
$\mathit{braun\_indices}\ \langle l,\ \_,\ r\rangle$
$= \{1\} \cup (\lambda i.\ i \cdot 2)\ `\ \mathit{braun\_indices}\ l\ \cup$
$\quad (\lambda i.\ i \cdot 2 + 1)\ `\ \mathit{braun\_indices}\ r$

The *braun_indices* of a tree are the numbers for which *lookup*1 (Section 3.2) is defined. Our main result is that Braun trees are exactly the trees that encode arrays:

Theorem 6.1. $\mathit{braun}\ t \longleftrightarrow \mathit{braun\_indices}\ t = \{1..|t|\}$

We start with some auxiliary properties:

$$(\lambda i.\ i \cdot 2)\ `\ \{a..b\} \cup (\lambda i.\ i \cdot 2 + 1)\ `\ \{a..b\}$$
$$= \{2 \cdot a..2 \cdot b + 1\} \tag{21}$$

$$S = \{m..n\} \cap \{i \mid \mathit{even}\ i\} \longrightarrow$$
$$(\exists\, m'\, n'.\ S = (\lambda i.\ i \cdot 2)\ `\ \{m'..n'\}) \tag{22}$$

$$S = \{m..n\} \cap \{i \mid \mathit{odd}\ i\} \longrightarrow$$
$$(\exists\, m'\, n'.\ S = (\lambda i.\ i \cdot 2 + 1)\ `\ \{m'..n'\}) \tag{23}$$

These proofs are all mostly automatic. We then show that the size (cardinality *card* in Isabelle) of the index set agrees with the size of the tree:

Lemma 6.2. $\mathit{card}\ (\mathit{braun\_indices}\ t) = |t|$

Proof. By induction. In the inductive step, $t = \langle l,\ x,\ r\rangle$, the index set of t is the three way union seen in the definition of *braun_indices*. We prove additional lemmas that show that the unions are disjoint and that the images apply injective functions, and the goal follows. □

We can now show that, if the index set is an interval, it is the expected one:

$$\mathit{braun\_indices}\ t = \{m..n\} \longrightarrow \{m..n\} = \{1..|t|\} \tag{24}$$

It is easy to show the lower bound must be 1, and the known cardinality tells us the upper bound.

The two directions of Theorem 6.1 are proved separately:

Lemma 6.3. $\mathit{braun}\ t \longrightarrow \mathit{braun\_indices}\ t = \{1..|t|\}$

Proof. The proof is by induction on t. In the inductive step, $t = \langle l,\ x,\ r\rangle$, the subtrees l and r must also be Braun trees, and the induction hypotheses tell us that their index sets form intervals $\{1..|l|\}$ and $\{1..|r|\}$. The sizes must also satisfy the usual constraints. The *braun_indices* of t are combined from these two intervals and the additional element 1. Lemma (21) shows we can merge these intervals, and the proof is completed with some special-case reasoning about 1 and an optional last element which exists if l is larger than r. □

Lemma 6.4. $\mathit{braun\_indices}\ t = \{1..|t|\} \longrightarrow \mathit{braun}\ t$

Proof. By induction, focusing on the inductive step where $t = \langle l,\ x,\ r\rangle$, with the premise that $\mathit{braun\_indices}\ t = \{1..|t|\}$. We can specialise that premise to the odd and even subsets, eliminate 1 as a special case, and derive a pair of equalities:

$$(\lambda i.\ i \cdot 2)\ `\ \mathit{braun\_indices}\ l = \{2..|t|\} \cap \{i \mid \mathit{even}\ i\} \tag{*}$$
$$(\lambda i.\ i \cdot 2 + 1)\ `\ \mathit{braun\_indices}\ r = \{2..|t|\} \cap \{i \mid \mathit{odd}\ i\} \tag{**}$$

We can now prove the index sets of the subtrees are intervals. We prove $\mathit{braun\_indices}\ l = \{1..|l|\}$ (from (*) and (22), (24)) and $\mathit{braun\_indices}\ r = \{1..|r|\}$ (from (**), (23) and (24)). These are the premises of the induction hypotheses, giving us *braun* l and *braun* r.

The complicated part is to prove the Braun size constraints. We know that $|t|$ must be a member of the LHS sets of (*) and (**) if it is a member of the RHS set, and likewise for $|t| - 1$. This gives us four implications. From these four implications and the interval properties, and by considering various parity cases, Isabelle can automatically show the Braun size constraints $|l| = |r| \lor |l| = |r| + 1$. □

7 Priority Queues via Braun Trees

Another application of Braun trees is to implement priority queues. Maintaining the Braun shape invariant is a simple way to ensure logarithmic depth.

Paulson [16] presents such an implementation (which he credits to Okasaki). Here we show that implementation and expand on our correctness proof. The Isabelle sources are available in the Archive of Formal Proofs [10].

This is the first verification of Paulson's implementation. Filliâtre has verified a slightly different version which is available in the Why3 gallery of verified programs [5]. For completeness we also present that version in Section 7.1 below.

The priority queue is another ADT interface, defined abstractly for types $'a$ with a linear order. The operations are:

$\mathit{insert} :: {'a} \Rightarrow {'q} \Rightarrow {'q}$
$\mathit{get\_min} :: {'q} \Rightarrow {'a}$ $\qquad \mathit{del\_min} :: {'q} \Rightarrow {'q}$
$\mathit{empty} :: {'q}$ $\qquad \mathit{is\_empty} :: {'q} \Rightarrow \mathit{bool}$

The abstract operations are specified using multisets [12]. We will use the function *mset*, which converts a list into a multiset, and *mset_tree*, which does the same for trees. The singleton multiset is denoted $\{\#x\#\}$, and multisets also support addition and subtraction.

We omit the ADT full specification and focus on the key operations, insertion and minimum-deletion. The array operations on Braun trees are not useful in this setting.

The implementation uses trees with a stronger invariant. They have Braun shape, and the elements are also ordered as a heap:

$$\begin{array}{l}
\mathit{heap} :: \mathit{'a\ tree} \Rightarrow \mathit{bool}\\
\mathit{heap}\ \langle\rangle = \mathit{True}\\
\mathit{heap}\ \langle l,\ m,\ r\rangle\\
= (\mathit{heap}\ l \wedge \mathit{heap}\ r \wedge (\forall\, x{\in}\mathit{set\_tree}\ l \cup \mathit{set\_tree}\ r.\ m \le x))
\end{array}$$

Insertion into a heap and Braun tree is by simple recursion:

$$\begin{array}{l}
\mathit{insert} :: \mathit{'a} \Rightarrow \mathit{'a\ tree} \Rightarrow \mathit{'a\ tree}\\
\mathit{insert}\ a\ \langle\rangle = \langle\langle\rangle,\ a,\ \langle\rangle\rangle\\
\mathit{insert}\ a\ \langle l,\ x,\ r\rangle\\
= (\text{if } a < x \text{ then } \langle \mathit{insert}\ x\ r,\ a,\ l\rangle \text{ else } \langle \mathit{insert}\ a\ r,\ x,\ l\rangle)
\end{array}$$

The key properties of insertion are straightforward to prove:

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{insert}\ x\ t)$$

$$\mathit{heap}\ t \longrightarrow \mathit{heap}\ (\mathit{insert}\ x\ t)$$

$$\mathit{mset\_tree}\ (\mathit{insert}\ x\ t) = \{\#x\#\} + \mathit{mset\_tree}\ t$$

$$|\mathit{insert}\ x\ t| = |t| + 1$$

The difficult operation is deletion of the minimum element from the root of the tree, leaving two subtrees to be merged. This is performed by two recursive functions, one to extract the leftmost element from a tree, and another to reassemble the heap. We have reproduced Paulson's definition of these functions almost verbatim; only the base case of *del_left* has been tuned slightly. These functions are specialised to Braun trees, with some cases missing (unspecified) that are impossible in the case of a Braun tree.

$$\begin{array}{l}
\mathit{del\_left} :: \mathit{'a\ tree} \Rightarrow \mathit{'a} \times \mathit{'a\ tree}\\
\mathit{del\_left}\ \langle\langle\rangle,\ x,\ r\rangle = (x,\ r)\\
\mathit{del\_left}\ \langle l,\ x,\ r\rangle = (\text{let } (y,\ l') = \mathit{del\_left}\ l \text{ in } (y,\ \langle r,\ x,\ l'\rangle))
\end{array}$$

$$\begin{array}{l}
\mathit{sift\_down} :: \mathit{'a\ tree} \Rightarrow \mathit{'a} \Rightarrow \mathit{'a\ tree} \Rightarrow \mathit{'a\ tree}\\
\mathit{sift\_down}\ \langle\rangle\ a\ \mathit{uu} = \langle\langle\rangle,\ a,\ \langle\rangle\rangle\\
\mathit{sift\_down}\ \langle\langle\rangle,\ x,\ \mathit{uv}\rangle\ a\ \langle\rangle\\
= (\text{if } a \le x\\
\quad \text{then } \langle\langle\langle\rangle,\ x,\ \langle\rangle\rangle,\ a,\ \langle\rangle\rangle\\
\quad \text{else } \langle\langle\langle\rangle,\ a,\ \langle\rangle\rangle,\ x,\ \langle\rangle\rangle)\\
\mathit{sift\_down}\ \langle l_1,\ x_1,\ r_1\rangle\ a\ \langle l_2,\ x_2,\ r_2\rangle\\
= (\text{if } a \le x_1 \wedge a \le x_2 \text{ then } \langle\langle l_1,\ x_1,\ r_1\rangle,\ a,\ \langle l_2,\ x_2,\ r_2\rangle\rangle\\
\quad \text{else if } x_1 \le x_2\\
\qquad \text{then } \langle \mathit{sift\_down}\ l_1\ a\ r_1,\ x_1,\ \langle l_2,\ x_2,\ r_2\rangle\rangle\\
\qquad \text{else } \langle\langle l_1,\ x_1,\ r_1\rangle,\ x_2,\ \mathit{sift\_down}\ l_2\ a\ r_2\rangle)
\end{array}$$

The deletion operation combines *del_left* and *sift_down*:

$$\begin{array}{l}
\mathit{del\_min} :: \mathit{'q} \Rightarrow \mathit{'q}\\
\mathit{del\_min}\ \langle\rangle = \langle\rangle\\
\mathit{del\_min}\ \langle\langle\rangle,\ x,\ r\rangle = \langle\rangle\\
\mathit{del\_min}\ \langle l,\ x,\ r\rangle = (\text{let } (y,\ l') = \mathit{del\_left}\ l \text{ in } \mathit{sift\_down}\ r\ y\ l')
\end{array}$$

The correctness properties of *del_left* are shown in a sequence of inductive proofs:

$$\mathit{del\_left}\ t = (x,\ t') \wedge t \ne \langle\rangle \longrightarrow |t| = |t'| + 1 \tag{25}$$

$$\mathit{del\_left}\ t = (x,\ t') \wedge \mathit{braun}\ t \wedge t \ne \langle\rangle \longrightarrow \mathit{braun}\ t' \tag{26}$$

$$\begin{array}{l}\mathit{del\_left}\ t = (x,\ t') \wedge t \ne \langle\rangle \longrightarrow\\ \mathit{set\_tree}\ t = \{x\} \cup \mathit{set\_tree}\ t'\end{array} \tag{27}$$

$$\mathit{del\_left}\ t = (x,\ t') \wedge t \ne \langle\rangle \wedge \mathit{heap}\ t \longrightarrow \mathit{heap}\ t' \tag{28}$$

$$\begin{array}{l}\mathit{del\_left}\ t = (x,\ t') \wedge t \ne \langle\rangle \longrightarrow\\ \mathit{mset\_tree}\ t = \{\#x\#\} + \mathit{mset\_tree}\ t'\end{array} \tag{29}$$

$$\begin{array}{l}\mathit{del\_left}\ t = (x,\ t') \wedge t \ne \langle\rangle \longrightarrow\\ x \in\#\ \mathit{mset\_tree}\ t \wedge \mathit{mset\_tree}\ t' = \mathit{mset\_tree}\ t - \{\#x\#\}\end{array} \tag{30}$$

Each of the key properties above requires an auxiliary lemma. We use a fact about tree size (25) to show the Braun size invariants (26) and likewise a lemma about the tree contents (27) to show the heap property (28). It is convenient to prove a multiset addition property (29) by induction and derive the expected multiset subtraction property (30). Multiset addition has convenient algebraic properties, but subtraction requires side conditions about whether we subtract more elements than were present.

The correctness properties of *sift_down* are also proved as a chain of simple inductive proofs. Again the Braun and heap properties are supported by lemmas about tree size and contents. Each lemma assumes the input is a Braun tree, as the function is not fully specified in other cases.

$$\mathit{braun}\ \langle l,\ a,\ r\rangle \longrightarrow |\mathit{sift\_down}\ l\ a\ r| = |l| + |r| + 1$$

$$\mathit{braun}\ \langle l,\ a,\ r\rangle \longrightarrow \mathit{braun}\ (\mathit{sift\_down}\ l\ a\ r)$$

$$\begin{array}{l}\mathit{braun}\ \langle l,\ a,\ r\rangle \longrightarrow\\ \mathit{set\_tree}\ (\mathit{sift\_down}\ l\ a\ r) = \{a\} \cup (\mathit{set\_tree}\ l \cup \mathit{set\_tree}\ r)\end{array}$$

$$\mathit{braun}\ \langle l,\ a,\ r\rangle \wedge \mathit{heap}\ l \wedge \mathit{heap}\ r \longrightarrow \mathit{heap}\ (\mathit{sift\_down}\ l\ a\ r)$$

$$\begin{array}{l}\mathit{braun}\ \langle l,\ a,\ r\rangle \longrightarrow\\ \mathit{mset\_tree}\ (\mathit{sift\_down}\ l\ a\ r)\\ = \{\#a\#\} + (\mathit{mset\_tree}\ l + \mathit{mset\_tree}\ r)\end{array}$$

The essential results about *del_min* follow:

$$\mathit{braun}\ t \longrightarrow \mathit{braun}\ (\mathit{del\_min}\ t)$$

$$\mathit{heap}\ t \wedge \mathit{braun}\ t \longrightarrow \mathit{heap}\ (\mathit{del\_min}\ t)$$

$$\begin{array}{l}\mathit{braun}\ t \wedge t \ne \langle\rangle \longrightarrow\\ \mathit{mset\_tree}\ (\mathit{del\_min}\ t) = \mathit{mset\_tree}\ t - \{\#\mathit{value}\ t\#\}\end{array}$$

7.1 A Variant of *del_min*

Filliâtre's counterpart to *del_min*, which we call *del_min*2 below (and which is called *remove_min* in [5]) combines the two subtrees below the root via a binary *merge* function instead of the ternary *sift_down*. During merging, if the root value of the right tree is moved up, preservation of the Braun invariant requires that it is replaced by an element from the left tree. This is the complete definition:

$$\begin{array}{l}
\mathit{le\_root} :: {'a} \Rightarrow {'a}\ \mathit{tree} \Rightarrow \mathit{bool}\\
\mathit{le\_root}\ a\ t = (t = \langle\rangle \vee a \le \mathit{value}\ t)
\end{array}$$

$$\begin{array}{l}
\mathit{replace\_min} :: {'a} \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}\\
\mathit{replace\_min}\ x\ \langle l,\ \_,\ r\rangle\\
= (\text{if } \mathit{le\_root}\ x\ l \wedge \mathit{le\_root}\ x\ r \text{ then } \langle l,\ x,\ r\rangle\\
\quad \text{else let } a = \mathit{value}\ l\\
\qquad\quad \text{in if } \mathit{le\_root}\ a\ r \text{ then } \langle \mathit{replace\_min}\ x\ l,\ a,\ r\rangle\\
\qquad\qquad \text{else } \langle l,\ \mathit{value}\ r,\ \mathit{replace\_min}\ x\ r\rangle)
\end{array}$$

$$\begin{array}{l}
\mathit{merge} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}\\
\mathit{merge}\ l\ \langle\rangle = l\\
\mathit{merge}\ \langle l_1,\ a_1,\ r_1\rangle\ \langle l_2,\ a_2,\ r_2\rangle\\
= (\text{if } a_1 \le a_2 \text{ then } \langle\langle l_2,\ a_2,\ r_2\rangle,\ a_1,\ \mathit{merge}\ l_1\ r_1\rangle\\
\quad \text{else let } (x,\ l') = \mathit{del\_left}\ \langle l_1,\ a_1,\ r_1\rangle\\
\qquad\quad \text{in } \langle \mathit{replace\_min}\ x\ \langle l_2,\ a_2,\ r_2\rangle,\ a_2,\ l'\rangle)
\end{array}$$

$$\begin{array}{l}
\mathit{del\_min2} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}\\
\mathit{del\_min2}\ \langle\rangle = \langle\rangle\\
\mathit{del\_min2}\ \langle l,\ \_,\ r\rangle = \mathit{merge}\ l\ r
\end{array}$$

It turns out that *replace_min* is just *sift_down* in disguise:

$$\mathit{braun}\ \langle l,\ a,\ r\rangle \longrightarrow \mathit{replace\_min}\ x\ \langle l,\ a,\ r\rangle = \mathit{sift\_down}\ l\ x\ r$$

This means that *del_min*2 is merely a slight optimization of *del_min*: instead of calling *del_left* right away, *merge* can take advantage of the case where the smaller element is at the root of the left heap and can be moved up without complications. However, on average this is just the case on the first level.

The correctness properties for *del_min*2 are the same as for *del_min* (see above) and follow easily from the lemmas about *sift_down* above (because *replace_min* is just *sift_down*) and these inductive lemmas about *merge*:

$$\begin{array}{l}
\mathit{braun}\ \langle l,\ x,\ r\rangle \longrightarrow\\
\mathit{mset\_tree}\ (\mathit{merge}\ l\ r) = \mathit{mset\_tree}\ l + \mathit{mset\_tree}\ r
\end{array}$$

$$\mathit{braun}\ \langle l,\ x,\ r\rangle \wedge \mathit{heap}\ l \wedge \mathit{heap}\ r \longrightarrow \mathit{heap}\ (\mathit{merge}\ l\ r)$$

$$\mathit{braun}\ \langle l,\ x,\ r\rangle \longrightarrow \mathit{braun}\ (\mathit{merge}\ l\ r) \wedge |\mathit{merge}\ l\ r| = |l| + |r|$$

The proofs can be found online [10].

8 Sorting via Priority Queues

One immediate application of a priority queue is to provide a sort operation. A list can be sorted by pushing its elements into the queue and retrieving them in order. Implementations of sorting using Braun trees have been presented in the literature by Paulson [16] and also by Guttman *et al.* [7]. Paulson's code does not come with proofs; Guttmann *et al.* derive their algorithm from a specification by program transformations but do not address time complexity. We verify both approaches, including the proof that their time complexity is $O(n \cdot \lg n)$.

Algorithm A, by Guttmann *et al.*, constructs a heap as described above, by inserting every element of a list:

$$\begin{array}{l}
\mathit{heap\_of}_A :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{tree}\\
\mathit{heap\_of}_A\ [] = \langle\rangle\\
\mathit{heap\_of}_A\ (a\ \#\ \mathit{as}) = \mathit{insert}\ a\ (\mathit{heap\_of}_A\ \mathit{as})
\end{array}$$

Algorithm B, by Paulson, constructs a heap differently, constructing a collection of heaps in a similar manner to an array-based heap sort:

$$\begin{array}{l}
\mathit{heapify} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{tree} \times {'a}\ \mathit{list}\\
\mathit{heapify}\ 0\ \mathit{xs} = (\langle\rangle,\ \mathit{xs})\\
\mathit{heapify}\ (n + 1)\ (x\ \#\ \mathit{xs})\\
= (\text{let } (l,\ \mathit{ys}) = \mathit{heapify}\ ((n + 1)\ \mathit{div}\ 2)\ \mathit{xs};\\
\qquad\quad (r,\ \mathit{zs}) = \mathit{heapify}\ (n\ \mathit{div}\ 2)\ \mathit{ys}\\
\quad\ \ \text{in } (\mathit{sift\_down}\ l\ x\ r,\ \mathit{zs}))
\end{array}$$

$$\begin{array}{l}
\mathit{heap\_of}_B :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{tree}\\
\mathit{heap\_of}_B\ \mathit{xs} = \mathit{fst}\ (\mathit{heapify}\ |\mathit{xs}|\ \mathit{xs})
\end{array}$$

The correctness properties of $\mathit{heap\_of}_A$ are easy to prove given the correctness of *insert*:

$$\mathit{heap}\ (\mathit{heap\_of}_A\ \mathit{xs})$$

$$\mathit{braun}\ (\mathit{heap\_of}_A\ \mathit{xs})$$

$$\mathit{mset\_tree}\ (\mathit{heap\_of}_A\ \mathit{xs}) = \mathit{mset}\ \mathit{xs}$$

The correctness of *heapify* is more complicated. Firstly we prove an auxiliary lemma about the remainder element returned by *heapify*:

$$\mathit{heapify}\ n\ \mathit{xs} = (t,\ \mathit{ys}) \wedge n \le |\mathit{xs}| \longrightarrow \mathit{ys} = \mathit{drop}\ n\ \mathit{xs}$$

We then state a single correctness theorem for proof by induction:

$$\begin{array}{l}
n \le |\mathit{xs}| \wedge \mathit{heapify}\ n\ \mathit{xs} = (t,\ \mathit{ys}) \longrightarrow\\
|t| = n \wedge \mathit{heap}\ t \wedge \mathit{braun}\ t \wedge \mathit{mset\_tree}\ t = \mathit{mset}\ (\mathit{take}\ n\ \mathit{xs})
\end{array}$$

The induction is on the recursion pattern of *heapify*. The proof follows from the correctness properties for *sift_down*. The proof is conceptually straightforward, but complicated by side conditions about division. The proof also requires hand instantiation of the following fact about *take* and *drop*: *drop*:

$$\mathit{mset}\ (\mathit{take}\ n\ \mathit{xs}) + \mathit{mset}\ (\mathit{drop}\ n\ \mathit{xs}) = \mathit{mset}\ \mathit{xs}$$

The above lemma is instantiated to show that the multisets generated by the two recursive calls can be merged, since one is essentially a *take* of the early elements and the other taken from list with those elements dropped.

Algorithm A reduces a heap to a list using a merge operation:

$$\begin{array}{l}
\mathit{merge} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}\\
\mathit{merge}\ \langle\rangle\ t_2 = t_2\\
\mathit{merge}\ t_1\ \langle\rangle = t_1\\
\mathit{merge}\ \langle l_1,\ a_1,\ r_1\rangle\ \langle l_2,\ a_2,\ r_2\rangle\\
= (\text{if } a_1 \le a_2\\
\quad\ \ \text{then } \langle \mathit{merge}\ l_1\ r_1,\ a_1,\ \langle l_2,\ a_2,\ r_2\rangle\rangle\\
\quad\ \ \text{else } \langle\langle l_1,\ a_1,\ r_1\rangle,\ a_2,\ \mathit{merge}\ l_2\ r_2\rangle)
\end{array}$$

$$\begin{array}{l}
\mathit{list\_of}_A :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{list}\\
\mathit{list\_of}_A\ \langle\rangle = []\\
\mathit{list\_of}_A\ \langle l,\ a,\ r\rangle = a\ \#\ \mathit{list\_of}_A\ (\mathit{merge}\ l\ r)
\end{array}$$

Algorithm B uses the *del_min* operation of the priority queue ADT:

$\mathit{list\_of}_B :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{list}$
$\mathit{list\_of}_B\ \langle\rangle = []$
$\mathit{list\_of}_B\ \langle l, a, r\rangle = a\ \#\ \mathit{list\_of}_B\ (\mathit{del\_min}\ \langle l, a, r\rangle)$

The interesting aspect of the approach of Guttman *et al.* is that the *merge* operation does not preserve the Braun shape invariant, in contrast to the *merge* function in Section 7.1. This makes the code simple but allows the tree to become unbalanced.

The correctness properties for $\mathit{list\_of}_A$ build on those of *merge*, and all are straightforward to prove:

$\mathit{mset\_tree}\ (\mathit{merge}\ l\ r) = \mathit{mset\_tree}\ l + \mathit{mset\_tree}\ r$

$\mathit{set\_tree}\ (\mathit{merge}\ l\ r) = \mathit{set\_tree}\ l \cup \mathit{set\_tree}\ r$

$\mathit{heap}\ l \wedge \mathit{heap}\ r \longrightarrow \mathit{heap}\ (\mathit{merge}\ l\ r)$

$\mathit{mset}\ (\mathit{list\_of}_A\ t) = \mathit{mset\_tree}\ t$

$\mathit{set}\ (\mathit{list\_of}_A\ t) = \mathit{set\_tree}\ t$

$\mathit{heap}\ t \longrightarrow \mathit{sorted}\ (\mathit{list\_of}_A\ t)$

We are interested in the multiset properties, but must also show the set properties since the *heap* and *sorted* predicates are defined in terms of those.

Together with the correctness result for $\mathit{heap\_of}_A$, this proves the functional correctness of algorithm A:

$\mathit{sorted}\ (\mathit{list\_of}_A\ (\mathit{heap\_of}_A\ \mathit{xs}))$

$\mathit{mset}\ (\mathit{list\_of}_A\ (\mathit{heap\_of}_A\ \mathit{xs})) = \mathit{mset}\ \mathit{xs}$

The correctness of $\mathit{list\_of}_B$ follows from the correctness of *del_min*. This proof requires us to address a technical detail: *del_min* calls *sift_down* and *sift_down* is partly underspecified. The *sift_down* implementation ignores the case where the right subtree is populated and the left subtree empty, since that is impossible for Braun trees. It would be possible to extend *sift_down* into a total function, but it requires multiple additional (redundant) cases and means a substantial change from Paulson's presentation. However the underspecification prevents us proving termination of $\mathit{list\_of}_B$ in general. Instead we prove that $\mathit{list\_of}_B$ is terminating for all input Braun trees, which is true as the tree size $|t|$ decreases for each recursion.

The correctness of $\mathit{list\_of}_B$ (for Braun trees) can then be shown by measure induction on $|t|$:

$\mathit{braun}\ t \longrightarrow \mathit{mset}\ (\mathit{list\_of}_B\ t) = \mathit{mset\_tree}\ t$

$\mathit{braun}\ t \longrightarrow \mathit{set}\ (\mathit{list\_of}_B\ t) = \mathit{set\_tree}\ t$

$\mathit{braun}\ t \wedge \mathit{heap}\ t \longrightarrow \mathit{sorted}\ (\mathit{list\_of}_B\ t)$

This establishes the functional correctness of algorithm B:

$\mathit{sorted}\ (\mathit{list\_of}_B\ (\mathit{heap\_of}_B\ \mathit{xs}))$

$\mathit{mset}\ (\mathit{list\_of}_B\ (\mathit{heap\_of}_B\ \mathit{xs})) = \mathit{mset}\ \mathit{xs}$

8.1 Running Time Analysis

Again we define a 'time' function for each function of interest. These are the time functions required to analyse *insert* (see Section 7) and $\mathit{heap\_of}_A$, the heap-creation part of algorithm A:

$\mathit{t\_insert} :: {'a} \Rightarrow {'a}\ \mathit{tree} \Rightarrow \mathit{nat}$
$\mathit{t\_insert}\ \_\ \langle\rangle = 1$
$\mathit{t\_insert}\ a\ \langle\_, x, r\rangle$
$= (\text{if } a < x \text{ then } 1 + \mathit{t\_insert}\ x\ r \text{ else } 1 + \mathit{t\_insert}\ a\ r)$

$\mathit{t\_heap\_of}_A :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$
$\mathit{t\_heap\_of}_A\ [] = 0$
$\mathit{t\_heap\_of}_A\ (a\ \#\ \mathit{as})$
$= \mathit{t\_insert}\ a\ (\mathit{heap\_of}_A\ \mathit{as}) + \mathit{t\_heap\_of}_A\ \mathit{as}$

The time functions we use in this section count the number of new constructor cells required for each function. The exception will be the list length function which requires no new constructors but which we will use as a time function for itself.

In the following proofs, we will mostly use the height of the Braun tree $h\ t$ as a proxy for the logarithm of the size of the tree. We proved before (in Section 2.1) that Braun trees have logarithmic size. We can usually reason directly about the effect of the operations on the size of the tree and avoid reasoning about logarithms.

We prove that $\mathit{heap\_of}_A$ has complexity $O(n \cdot \lg n)$ using a chain of lemmas:

$\mathit{t\_insert}\ x\ t \leq h\ t + 1$

$h\ t \leq h\ (\mathit{insert}\ x\ t)$

$\mathit{t\_heap\_of}_A\ \mathit{xs} \leq |\mathit{xs}| \cdot (h\ (\mathit{heap\_of}_A\ \mathit{xs}) + 1)$

It is more interesting and challenging to analyse the heap construction of algorithm B. These are the time functions needed (list length is used to time itself):

$\mathit{t\_heapify} :: \mathit{nat} \Rightarrow {'a}\ \mathit{list} \Rightarrow \mathit{nat}$
$\mathit{t\_heapify}\ 0\ \_ = 1$
$\mathit{t\_heapify}\ (n + 1)\ (x\ \#\ \mathit{xs})$
$= (\text{let } (l, \mathit{ys}) = \mathit{heapify}\ ((n + 1)\ \mathit{div}\ 2)\ \mathit{xs};$
$\quad\quad t_1 = \mathit{t\_heapify}\ ((n + 1)\ \mathit{div}\ 2)\ \mathit{xs};$
$\quad\quad (r, \mathit{zs}) = \mathit{heapify}\ (n\ \mathit{div}\ 2)\ \mathit{ys};$
$\quad\quad t_2 = \mathit{t\_heapify}\ (n\ \mathit{div}\ 2)\ \mathit{ys}$
$\quad \text{in } 1 + t_1 + t_2 + \mathit{t\_sift\_down}\ l\ x\ r)$

$\mathit{t\_heap\_of}_B :: {'a}\ \mathit{list} \Rightarrow \mathit{nat}$
$\mathit{t\_heap\_of}_B\ \mathit{xs} = |\mathit{xs}| + \mathit{t\_heapify}\ |\mathit{xs}|\ \mathit{xs}$

We can prove that *heapify* has linear time complexity. Because *heapify* is a divide and conquer algorithm, we can in principle determine its asymptotic complexity using the "master theorem" [2]. However, a verified master theorem is a nontrivial undertaking and it appears that it is currently only available in Isabelle [4]. To make our proof pearl self contained we give a direct proof.

We begin with two properties about *sift_down* and tree height, both of which are easy to prove:

$$braun\ \langle l, x, r\rangle \longrightarrow h\ (sift\_down\ l\ x\ r) \le h\ \langle l, x, r\rangle \qquad (31)$$

$$braun\ \langle l, x, r\rangle \longrightarrow t\_sift\_down\ l\ x\ r \le h\ \langle l, x, r\rangle \qquad (32)$$

This key lemma implies an $O(n)$ complexity of *heapify*:

$$i \le |xs| \longrightarrow t\_heapify\ i\ xs + h\ (fst\ (heapify\ i\ xs)) \le 5 \cdot i + 1$$

The proof is by induction on the recursion of *t_heapify*. The challenging part is the inductive step. We must prove an inequality from two inductive hypotheses, a problem with the following form:

$$t_1 + h\ l \le \ldots \longrightarrow$$
$$t_2 + h\ r \le \ldots \longrightarrow$$
$$1 + t_1 + t_2 + t\_sift\_down\ l\ x\ r + h\ (sift\_down\ l\ x\ r) \le \ldots$$

The times t_1, t_2 and variables l, x, r are from the definition of *t_heapify* above. The lemma was carefully phrased with an additional height term so that if we add together the two inequalities from the inductive premises, we get a new inequality with a very similar shape to the one we must prove.

Isabelle can prove the inductive goal from the sum inequality if we first establish these properties:

$$t\_sift\_down\ l\ x\ r \le h\ l + 1$$
$$h\ (sift\_down\ l\ x\ r) \le h\ r + 2$$

To establish these subgoals within our inductive case, we repeat the proof of *braun* $\langle l, x, r\rangle$ from the correctness proof of *heapify*. We can then use balance lemmas 2.5 and 2.6 to relate $h\ l$ and $h\ r$, which together with the height bounds (31) and (32) establish our subgoals. This completes the inductive proof and shows *heapify* has linear time complexity.

The complexity proofs about extracting lists from heaps are simpler. For algorithm A, we need these time functions:

$$t\_merge :: \ 'a\ tree \Rightarrow\ 'a\ tree \Rightarrow nat$$
$$t\_merge\ \langle\rangle\ \_ = 0$$
$$t\_merge\ \langle\_, \_, \_\rangle\ \langle\rangle = 0$$
$$t\_merge\ \langle l_1, a_1, r_1\rangle\ \langle l_2, a_2, r_2\rangle$$
$$= (\text{if } a_1 \le a_2 \text{ then } 1 + t\_merge\ l_1\ r_1 \text{ else } 1 + t\_merge\ l_2\ r_2)$$

$$t\_list\_of_A :: \ 'a\ tree \Rightarrow nat$$
$$t\_list\_of_A\ \langle\rangle = 0$$
$$t\_list\_of_A\ \langle l, \_, r\rangle = 1 + t\_merge\ l\ r + t\_list\_of_A\ (merge\ l\ r)$$

Firstly we show *merge* runs in time proportional to the height of the heap, which it cannot increase:

$$t\_merge\ l\ r \le max\ (h\ l)\ (h\ r)$$

$$h\ (merge\ l\ r) \le h\ \langle l, x, r\rangle$$

The time bound follows by induction:

$$t\_list\_of_A\ t \le 2 \cdot h\ t \cdot |t|$$

We can now convert heights to logarithms and prove the final timing result for algorithm A:

$$t\_heap\_of_A\ xs + t\_list\_of_A\ (heap\_of_A\ xs)$$
$$\le 3 \cdot |xs| \cdot (\lceil \lg\ (|xs| + 1)\rceil + 1)$$

For algorithm B, we have some more auxiliary constants to cover (*del_left* and *del_min* were defined in Section 7):

$$t\_del\_left :: \ 'a\ tree \Rightarrow nat$$
$$t\_del\_left\ \langle\langle\rangle, x, r\rangle = 1$$
$$t\_del\_left\ \langle l, x, r\rangle$$
$$= (\text{let } (y, l') = del\_left\ l \text{ in } 2 + t\_del\_left\ l)$$

$$t\_del\_min :: \ 'a\ tree \Rightarrow nat$$
$$t\_del\_min\ \langle\rangle = 0$$
$$t\_del\_min\ \langle\langle\rangle, x, r\rangle = 0$$
$$t\_del\_min\ \langle l, x, r\rangle$$
$$= (\text{let } (y, l') = del\_left\ l$$
$$\text{in } t\_del\_left\ l + t\_sift\_down\ r\ y\ l')$$

$$t\_list\_of_B :: \ 'a\ tree \Rightarrow nat$$
$$t\_list\_of_B\ \langle\rangle = 0$$
$$t\_list\_of_B\ \langle l, a, r\rangle$$
$$= 1 + t\_del\_min\ \langle l, a, r\rangle + t\_list\_of_B\ (del\_min\ \langle l, a, r\rangle)$$

We prove a chain of time and height bounds:

$$t \ne \langle\rangle \longrightarrow t\_del\_left\ t \le 2 \cdot h\ t$$

$$del\_left\ t = (v, t') \wedge t \ne \langle\rangle \longrightarrow h\ t' \le h\ t$$

$$braun\ t \longrightarrow t\_del\_min\ t \le 3 \cdot h\ t$$

$$braun\ t \longrightarrow h\ (del\_min\ t) \le h\ t$$

$$braun\ t \longrightarrow t\_list\_of_B\ t \le 3 \cdot (h\ t + 1) \cdot |t|$$

The proofs are all straightforward by induction, aside from the complication that *t_list_of*$_B$, like *list_of*$_B$, is only partially terminating, and we must prove again that Braun trees are in its termination domain.

The above results let us prove the total time for algorithm B is also $O(n \cdot \lg n)$:

$$t\_heap\_of_B\ xs + t\_list\_of_B\ (heap\_of_B\ xs)$$
$$\le 3 \cdot |xs| \cdot (\lceil \lg\ (|xs| + 1)\rceil + 3) + 1$$

9 Conclusion

We have thoroughly explored the topic of Braun trees, verifying all algorithms in Isabelle/HOL: flexible arrays, priority queues and sorting functions based on them. This includes the first correctness proofs of Okasaki's conversion from lists to Braun trees and the first presentation of a linear time conversion in the other direction. We have also presented a novel combinatorial characterization of Braun trees.

References

[1] Clemens Ballarin. *Tutorial to Locales and Locale Interpretation.* https://isabelle.in.tum.de/doc/locales.pdf.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 3rd edition, 2009.

[3] Rene De La Briandais. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 295–298, New York, NY, USA, 1959. ACM.

[4] Manuel Eberl. Proving divide and conquer complexities in Isabelle/HOL. *Journal of Automated Reasoning*, 58(4):483–508, 2017.

[5] Jean-Christophe Filliâtre. Purely applicative heaps implemented with Braun trees. *Gallery of Verified Programs*, 2015. http://toccata.lri.fr/gallery/braun_trees.en.html, Formal proof development.

[6] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[7] Walter Guttmann, Helmuth Partsch, Wolfram Schulte, and Ton Vullinghs. Tool support for the interactive derivation of formally correct functional programs. *Journal of Universal Computer Science*, 9(2):173–188, Feb 2003.

[8] Rob R. Hoogerwoord. A logarithmic implementation of flexible arrays. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction, Second International Conference*, volume 669 of *LNCS*, pages 191–207. Springer, 1992.

[9] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 2nd edition, 1990.

[10] Tobias Nipkow. Priority queues based on Braun trees. *Archive of Formal Proofs*, 2014. http://devel.isa-afp.org/entries/Priority_Queue_Braun.html (development) and http://isa-afp.org/entries/Priority_Queue_Braun.html (latest release), Formal proof development.

[11] Tobias Nipkow. Verified root-balanced trees. In Bor-Yuh Evan Chang, editor, *Asian Symposium on Programming Languages and Systems, APLAS 2017*, volume 10695 of *LNCS*, pages 255–272. Springer, 2017.

[12] Tobias Nipkow and Hauke Brinkop. Amortized complexity verified. *J. Automated Reasoning*, 62:367–391, 2019.

[13] Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. 298 pp. http://concrete-semantics.org.

[14] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 218 pp.

[15] Chris Okasaki. Three algorithms on Braun trees. *J. Functional Programming*, 7(6):661–666, 1997.

[16] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[17] Martin Rem and Wim Braun. A logarithmic implementation of flexible arrays. Memorandum MR83/4. Eindhoven University of Techology, 1983.

[18] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.