Jasmin Blanchette, Tobias Nipkow,
Christian Sternagel, Bohua Zhan

Verified Functional Data Structures and Algorithms

May 19, 2020

Contents

1

Basics

1.1 Basic Types

The syntax of types in Isabelle follows Standard ML [20]: **type variables** are denoted by *'a*, *'b* etc. Type operators follow their argument type(s), e.g. *'a list*. The notation $t :: \tau$ means that term t has type τ. The following types are predefined.

Type *bool* comes with the constants *True* and *False* and the usual operations.

There are three numeric types: the natural numbers *nat* (0, 1, ...), the integers *int* and the real numbers *real*. They correspond to the mathematical sets $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ and not to any machine arithmetic. All three types come with the usual (overloaded) operations. The type of natural numbers *nat* may be used in pattern-matching definitions, e.g. *fib* $(n + 2)$ = *fib* $(n + 1)$ + *fib* n.

The type *'a set* of sets (finite and infinite) over type *'a* comes with the standard mathematical operations. Note that set complement and difference is denoted by unary and binary $-$.

The type *'a list* of lists of elements of type *'a* is a recursive data type:

```
datatype 'a list = Nil | Cons 'a ('a list)
```

Constant [] represents the empty list and x # xs the list with first element x, the **head**, and rest list xs, the **tail**. The following syntactic sugar is sprinkled on top:

$$\begin{aligned} [] &\equiv \mathit{Nil} \\ x \# xs &\equiv \mathit{Cons}\ x\ xs \\ [x_1, \ldots, x_n] &\equiv x_1 \# \ldots \# x_n \# [] \end{aligned}$$

A library of predefined functions on lists is shown in Appendix A. The length of a list xs is denoted by $|xs|$.

The data type $'a$ *option* is defined as follows:

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

Pairs are written $(a,\ b)$. Functions *fst* and *snd* select the first and second component of a pair: *fst* $(a,\ b) = a$ and *snd* $(a,\ b) = b$. The type *unit* contains only a single element $()$, the empty tuple.

We will use well-known or obvious properties of the above predefined types implicitly without detailed justification, e.g. commutativity of $\cup$ or *set* (*filter* $P\ xs) = \{x \in$ *set* $xs \mid P\ x\}$.

1.1.1 Notation

We adopt the following notation:

- The type of functions $'a \Rightarrow\ 'b$ comes with a predefined pointwise update operation with its own notation:
  $f(a := b) = (\lambda x.\ \mathit{if}\ x = a\ \mathit{then}\ b\ \mathit{else}\ f\ x)$
- Any infix operator can be turned into a function by enclosing it in parentheses, e.g. $(+)$.
- Function lg is the binary logarithm.

Functions are defined by pattern matching similar to the programming language Haskell. In this book we occasionally use an extension of pattern matching where patterns can be named. For example, the defining equation

$f\ (x\ \#\ (y\ \#\ zs =: ys)) = ys\ @\ zs$

introduces a variables ys on the left that stands for $y\ \#\ zs$ and can be referred to on the right. Logically it is just an abbreviation of

$f\ (x\ \#\ y\ \#\ zs) = (\mathit{let}\ ys = y\ \#\ zs\ \mathit{in}\ ys\ @\ zs)$

although it is suggestive of a more efficient interpretation. The general format is *pattern* $=:$ *variable*.

We deviate from Isabelle's notation in favour of standard mathematics in a number of points:

- There is only one implication: $\Longrightarrow$ is printed as $\longrightarrow$ and $P \Longrightarrow Q \Longrightarrow R$ is printed as $P \wedge Q \longrightarrow R$.
- Multiplication is printed as $x \cdot y$.
- We sweep under the carpet that type *nat* is defined as a recursive data type: **datatype** *nat* $= 0 \mid$ *Suc nat*. In particular, constructor *Suc* is hidden: $Suc^{k}\ 0$ is printed as k and $Suc^{k}\ n$ (where n is not 0) is printed as $n + k$.

- Set comprehension syntax is the canonical $\{x \mid P\}$.

The reader who consults the Isabelle theories referred to in this book should beware of these discrepancies that improve readability.

1.1.2 Numeric Types and Coercions

The numeric types *nat*, *int* and *real* are all distinct and it requires explicit coercion functions to convert between them, in particular the embeddings *int* :: *nat* ⇒ *int* and *real* :: *nat* ⇒ *real*. We do not show embeddings unless it makes a difference. For example, $(m + n)$ :: *real*, where m, n :: *nat*, is mathematically unambiguous because *real* $(m + n) =$ *real* m + *real* n. On the other hand, $(m - n)$ :: *real* would be ambiguous because *real* $(m - n)$ $\neq$ *real* m − *real* n because subtraction on *nat* is cut off at 0. In some cases we can also drop coercion functions that are not embeddings, e.g. *nat* :: *int* ⇒ *nat*, which coerces negative integers to 0: if we know that $i \geqslant 0$ then we can drop the *nat* in *nat* i.

1.1.3 Multisets

Informally, a multiset is a set where elements can occur multiple times. Multisets come with the following operations:

$$\begin{aligned}
\{\#\} &:: {}'a\ \mathit{multiset} \\
(\in\#) &:: {}'a \Rightarrow {}'a\ \mathit{multiset} \Rightarrow \mathit{bool} \\
\mathit{add\_mset} &:: {}'a \Rightarrow {}'a\ \mathit{multiset} \Rightarrow {}'a\ \mathit{multiset} \\
(+) &:: {}'a\ \mathit{multiset} \Rightarrow {}'a\ \mathit{multiset} \Rightarrow {}'a\ \mathit{multiset} \\
\textstyle\bigcup\# &:: {}'a\ \mathit{multiset}\ \mathit{multiset} \Rightarrow {}'a\ \mathit{multiset} \\
\mathit{size} &:: {}'a\ \mathit{multiset} \Rightarrow \mathit{nat} \\
\mathit{mset} &:: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{multiset} \\
\mathit{set\_mset} &:: {}'a\ \mathit{multiset} \Rightarrow {}'a\ \mathit{set} \\
\mathit{image\_mset} &:: ({}'a \Rightarrow {}'b) \Rightarrow {}'a\ \mathit{multiset} \Rightarrow {}'b\ \mathit{multiset}
\end{aligned}$$

Their meaning: $\{\#\}$ is the empty multiset; $(\in\#)$ is the element test; *add_mset* adds an element to a multiset; $(+)$ is the union of two multisets, where multiplicities of elements are added; $\bigcup\#$ is the union of a multiset of multisets, the iteration of $(+)$; *size* M, written $|M|$, is the number of elements in M, taking multiplicities into account; *mset* converts a list into a multiset by forgetting about the order of elements; *set_mset* converts a multiset into a set; *image_mset* applies a function to all elements of a multiset.

Just as for arithmetic, lists and sets, we use obvious properties of multisets implicitly without detailed justification.

1.2 Running Time

Our approach to reasoning about the running time of a function f is very simple: we explicitly define a function t_f such that $t\_f\ x$ models the time the computation of $f\ x$ takes. Unless stated otherwise, t_f counts the number of all function calls In the computation of f. To simplify matters we sometimes count not all function calls but only specific ones, for example the number of comparisons when executing a sorting function.

TODO DETAILS

Initially we show the definition of each t_f. We stop doing so when the principles above have been exemplified often enough.

TODO Master Thm

Part I

Sorting and Selection

2 Sorting [1]

Sorting and searching frequently involves an ordering. In such cases in this book we simply assume that comparison operations $\leqslant$ and $<$ are defined on the underlying type. This is how sortedness of a list is defined:

$$\mathit{sorted} :: ('a{::}\mathit{linorder})\ \mathit{list} \Rightarrow \mathit{bool}$$
$$\mathit{sorted}\ [] = \mathit{True}$$
$$\mathit{sorted}\ (x \mathbin{\#} ys) = ((\forall y{\in}\mathit{set}\ ys.\ x \leqslant y) \wedge \mathit{sorted}\ ys)$$

That is, every element is $\leqslant$ than all elements to the right of it: the list is sorted in increasing order. But what does $'a{::}\mathit{linorder}$ mean?

The type variable $'a$ is annotated with *linorder*, which means that *sorted* is only applicable if a binary predicate $(\leqslant) :: 'a \Rightarrow 'a \Rightarrow \mathit{bool}$ is defined and $(\leqslant)$ is a **linear order**, i.e. the following properties are satisfied:

reflexivity:	$x \leqslant x$
transitivity:	$x \leqslant y \wedge y \leqslant z \longrightarrow x \leqslant z$
antisymmetry:	$x \leqslant y \wedge y \leqslant x \longrightarrow x = y$
linearity/totality:	$x \leqslant y \vee y \leqslant x$

Moreover, the binary predicate $(<)$ must satisfy

$$x < y \longleftrightarrow x \leqslant y \wedge x \neq y.$$

Note that *linorder* is a specific predefined example of a **type class** [14]. We will not explain type classes any further because we do not require the general concept. In fact, we will mostly do not even show the *linorder* annotation in types: you can assume that if you see a $\leqslant$ or $<$ on a generic type $'a$ in this book, $'a$ is implicitly annotated with *linorder*. Note further that $(\leqslant)$ on the numeric types *nat*, *int* and *real* is a linear order.

[1] `isabelle/src/HOL/Data_Structures/Sorting.thy`

2.1 Specification of Sorting Functions

A sorting function $\mathit{sort} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list}$ (where, as usual, $'a{::}\mathit{linorder}$) must obviously satisfy the following property:

$$\mathit{sorted}\ (\mathit{sort}\ \mathit{xs})$$

However, this is not enough — otherwise, $\mathit{wrong\_sort}\ \mathit{xs} = []$ would be a correct sorting function. The set of elements in the output must be the same as in the input, and each element must occur the same number of times This is most readily captured with the notion of a multiset (see Section 1.1.3). The second property that a sorting function *sort* must satisfy is

$$\mathit{mset}\ (\mathit{sort}\ \mathit{xs}) = \mathit{mset}\ \mathit{xs}$$

where function *mset* converts a list into its corresponding multiset.

2.2 Insertion Sort

$$\begin{array}{l}
\mathit{insort} :: {}'a \Rightarrow {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \\
\mathit{insort}\ x\ [] = [x] \\
\mathit{insort}\ x\ (y \# \mathit{ys}) = (\mathit{if}\ x \leqslant y\ \mathit{then}\ x \# y \# \mathit{ys}\ \mathit{else}\ y \# \mathit{insort}\ x\ \mathit{ys}) \\
\\
\mathit{isort} :: {}'a\ \mathit{list} \Rightarrow {}'a\ \mathit{list} \\
\mathit{isort}\ [] = [] \\
\mathit{isort}\ (x \# \mathit{xs}) = \mathit{insort}\ x\ (\mathit{isort}\ \mathit{xs})
\end{array}$$

2.2.1 Functional Correctness

We start by proving the preservation of the multiset of elements:

$$\mathit{mset}\ (\mathit{insort}\ x\ \mathit{xs}) = \mathit{add\_mset}\ x\ (\mathit{mset}\ \mathit{xs}) \tag{2.1}$$
$$\mathit{mset}\ (\mathit{isort}\ \mathit{xs}) = \mathit{mset}\ \mathit{xs} \tag{2.2}$$

Both are proved by induction; the proof of (2.2) requires (2.1).

Now we turn to sortedness. Because the definition of *sorted* involves *set*, it is frequently helpful to prove multiset preservation first (as we have done above) because that yields preservation of the set of elements. That is, from (2.1) we obtain

$$set\ (insort\ x\ xs) = insert\ x\ (set\ xs) \tag{2.3}$$

where *insert* inserts an element into a set. Two inductions prove

$$sorted\ (insort\ a\ xs) = sorted\ xs \tag{2.4}$$
$$sorted\ (isort\ xs) \tag{2.5}$$

where the proof of (2.4) uses (2.3) and the proof of (2.5) uses (2.4).

2.2.2 Running Time Analysis

We count function calls:

$$t\_insort :: 'a \Rightarrow 'a\ list \Rightarrow nat$$
$$t\_insort\ \_\ [] = 1$$
$$t\_insort\ x\ (y\ \#\ ys) = (if\ x \leqslant y\ then\ 0\ else\ t\_insort\ x\ ys) + 1$$

$$t\_isort :: 'a\ list \Rightarrow nat$$
$$t\_isort\ [] = 1$$
$$t\_isort\ (x\ \#\ xs) = t\_isort\ xs + t\_insort\ x\ (isort\ xs) + 1$$

Lemma 2.1. $t\_isort\ xs \leqslant (|xs| + 1)^2$

Proof. The following properties are proved by induction on xs:

$$t\_insort\ x\ xs \leqslant |xs| + 1 \tag{2.6}$$
$$|insort\ x\ xs| = |xs| + 1 \tag{2.7}$$
$$|isort\ xs| = |xs| \tag{2.8}$$

The proof of (2.8) needs (2.7). The proof of $t\_isort\ xs \leqslant (|xs| + 1)^2$ is also by induction on xs. The base case is trivial. The induction step is easy:

$$\begin{aligned}
&t\_isort\ (x\ \#\ xs) = t\_isort\ xs + t\_insort\ x\ (isort\ xs) + 1 \\
&\leqslant (|xs| + 1)^2 + t\_insort\ x\ (isort\ xs) + 1 && \text{by IH} \\
&\leqslant (|xs| + 1)^2 + |xs| + 1 + 1 && \text{using (2.6) and (2.8)} \\
&\leqslant (|x\ \#\ xs| + 1)^2 && \square
\end{aligned}$$

2.2.3 Exercises

Exercise 2.1. Show that any sorting function behaves like insertion sort:

$$(\forall xs.\ mset\ (f\ xs) = mset\ xs) \wedge (\forall xs.\ sorted\ (f\ xs)) \longrightarrow f\ xs = isort\ xs$$

2.3 Quicksort

$$
\begin{aligned}
&\mathit{quicksort} :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}\\
&\mathit{quicksort}\ [] = []\\
&\mathit{quicksort}\ (x \mathbin{\#} \mathit{xs})\\
&= \mathit{quicksort}\ (\mathit{filter}\ (\lambda y.\ y < x)\ \mathit{xs})\ @\\
&\quad [x]\ @\ \mathit{quicksort}\ (\mathit{filter}\ (\lambda y.\ x \leqslant y)\ \mathit{xs})
\end{aligned}
$$

2.3.1 Functional Correctness

Preservation of the multiset of elements

$$\mathit{mset}\ (\mathit{quicksort}\ \mathit{xs}) = \mathit{mset}\ \mathit{xs} \tag{2.9}$$

is proved by computation induction using the lemmas

$$
\begin{aligned}
&\mathit{mset}\ (\mathit{filter}\ P\ \mathit{xs}) = \mathit{filter\_mset}\ P\ (\mathit{mset}\ \mathit{xs})\\
&(\forall x.\ P\ x = (\neg\ Q\ x)) \longrightarrow \mathit{filter\_mset}\ P\ M + \mathit{filter\_mset}\ Q\ M = M
\end{aligned}
$$

where $\mathit{filter\_mset}\ P\ M$ is the multiset of elements in the multiset M that satisfy P.

A second computation induction proves sortedness

$$\mathit{sorted}\ (\mathit{quicksort}\ \mathit{xs})$$

using the lemmas

$$
\begin{aligned}
&\mathit{sorted}\ (\mathit{xs}\ @\ \mathit{ys})\\
&= (\mathit{sorted}\ \mathit{xs} \wedge \mathit{sorted}\ \mathit{ys} \wedge (\forall x{\in}\mathit{set}\ \mathit{xs}.\ \forall y{\in}\mathit{set}\ \mathit{ys}.\ x \leqslant y))\\
&\mathit{set}\ (\mathit{quicksort}\ \mathit{xs}) = \mathit{set}\ \mathit{xs}
\end{aligned}
$$

where the latter one is an easy consequence of (2.9).

We do not analyze the running time of *quicksort*. It is well known that in the worst case it is quadratic (exercise!) but that the expected running time is $n * \mathit{lg}\ n$. The necessary probabilistic analysis is beyond the scope of this text but can be found elsewhere [8, 9].

2.3.2 Exercises

Exercise 2.2. Function *quicksort* appends the lists returned from the recursive calls. This is expensive because the running time of (@) is linear in the length of its first argument. Define a function $\mathit{quicksort2} :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{list}$ that avoids (@) but accumulates the result in its second parameter via (#) only. Prove $\mathit{quicksort2}\ \mathit{xs}\ \mathit{ys} = \mathit{quicksort}\ \mathit{xs}\ @\ \mathit{ys}$.

2.4 Top-Down Merge Sort

```
merge :: 'a list ⇒ 'a list ⇒ 'a list
merge [] ys = ys
merge xs [] = xs
merge (x # xs) (y # ys)
= (if x ⩽ y then x # merge xs (y # ys) else y # merge (x # xs) ys)

msort :: 'a list ⇒ 'a list
msort xs
= (let n = |xs|
    in if n ⩽ 1 then xs
       else merge (msort (take (n div 2) xs))
              (msort (drop (n div 2) xs)))
```

2.4.1 Functional Correctness

We start off with multisets and sets of elements:

$$\mathit{mset}\ (\mathit{merge}\ xs\ ys) = \mathit{mset}\ xs + \mathit{mset}\ ys \tag{2.10}$$

$$\mathit{set}\ (\mathit{merge}\ xs\ ys) = \mathit{set}\ xs \cup \mathit{set}\ ys \tag{2.11}$$

Proposition (2.10) is proved by induction on the computation of *merge* and (2.11) is an easy consequence.

Lemma 2.2. $\mathit{mset}\ (\mathit{msort}\ xs) = \mathit{mset}\ xs$

Proof. The proof is by induction on the computation of *msort*. Let $n = |xs|$. The base case ($n \leqslant 1$) is trivial. Now assume $n > 1$ and let $ys = \mathit{take}\ (n\ \mathit{div}\ 2)\ xs$ and $zs = \mathit{drop}\ (n\ \mathit{div}\ 2)\ xs$.

$$\begin{aligned}
&\mathit{mset}\ (\mathit{msort}\ xs) = \mathit{mset}\ (\mathit{msort}\ ys) + \mathit{mset}\ (\mathit{msort}\ zs) && \text{by (2.10)}\\
&= \mathit{mset}\ ys + \mathit{mset}\ zs && \text{by IH}\\
&= \mathit{mset}\ (ys\ @\ zs)\\
&= \mathit{mset}\ xs && \square
\end{aligned}$$

Now we turn to sortedness. An induction on the computation of *merge*, using (2.11), yields

$$\mathit{sorted}\ (\mathit{merge}\ xs\ ys) = (\mathit{sorted}\ xs \wedge \mathit{sorted}\ ys) \tag{2.12}$$

Lemma 2.3. $\mathit{sorted}\ (\mathit{msort}\ xs)$

The proof is an easy induction on the computation of *msort*. The base case ($n \leqslant 1$) follows because every list of length $\leqslant 1$ is sorted. The induction step follows with the help of (2.12).

2.4.2 Running Time Analysis

To simplify the analysis, and in line with the literature, we only count the number of comparisons:

```
$c\_merge$ :: $'a\ list \Rightarrow 'a\ list \Rightarrow nat$
$c\_merge\ []\ ys = 0$
$c\_merge\ xs\ [] = 0$
$c\_merge\ (x\ \#\ xs)\ (y\ \#\ ys)$
$= 1 + (if\ x \leqslant y\ then\ c\_merge\ xs\ (y\ \#\ ys)\ else\ c\_merge\ (x\ \#\ xs)\ ys)$

$c\_msort$ :: $'a\ list \Rightarrow nat$
$c\_msort\ xs$
$= (let\ n = |xs|;\ ys = take\ (n\ div\ 2)\ xs;\ zs = drop\ (n\ div\ 2)\ xs$
    $in\ if\ n \leqslant 1\ then\ 0$
       $else\ c\_msort\ ys + c\_msort\ zs +$
            $c\_merge\ (msort\ ys)\ (msort\ zs))$
```

By computation inductions we obtain:

$$|merge\ xs\ ys| = |xs| + |ys| \tag{2.13}$$
$$|msort\ xs| = |xs| \tag{2.14}$$
$$c\_merge\ xs\ ys \leqslant |xs| + |ys| \tag{2.15}$$

where the proof of (2.14) uses (2.13).

To simplify technicalities we prove the "$n * lg\ n$" bound on the number of comparisons in *msort* only for $n = 2^k$, in which case the bound becomes $k \cdot 2^k$.

Lemma 2.4. $|xs| = 2^k \longrightarrow c\_msort\ xs \leqslant k \cdot 2^k$

Proof. The proof is by induction on k. The base case is trivial and we concentrate on the step. Let $n = |xs|$, $ys = take\ (n\ div\ 2)\ xs$ and $zs = drop\ (n\ div\ 2)\ xs$. The case $n \leqslant 1$ is trivial. Now assume $n > 1$.

$$\begin{aligned}
&c\_msort\ xs \\
&= c\_msort\ ys + c\_msort\ zs + c\_merge\ (msort\ ys)\ (msort\ zs) \\
&\leqslant c\_msort\ ys + c\_msort\ zs + |ys| + |zs| && \text{using (2.15) and (2.14)} \\
&\leqslant k \cdot 2^k + k \cdot 2^k + |ys| + |zs| && \text{by IH} \\
&= k \cdot 2^k + k \cdot 2^k + |xs| \\
&= (k + 1) \cdot 2^{k + 1} && \text{by assumption } |xs| = 2^{k + 1}
\end{aligned}$$ □

2.5 Bottom-Up Merge Sort

Bottom-up merge sort starts by turning the input $[x_1,...,x_n]$ into the list $[[x_1],...,[x_n]]$. Then it passes over this list of lists repeatedly, merging pairs of adjacent lists on every pass until at most one list is left.

```
merge_adj :: 'a list list ⇒ 'a list list
merge_adj [] = []
merge_adj [xs] = [xs]
merge_adj (xs # ys # zss) = merge xs ys # merge_adj zss

merge_all :: 'a list list ⇒ 'a list
merge_all [] = []
merge_all [xs] = xs
merge_all xss = merge_all (merge_adj xss)

msort_bu :: 'a list ⇒ 'a list
msort_bu xs = merge_all (map (λx. [x]) xs)
```

Termination of *merge_adj* relies on the fact that *merge_adj* halves the length of the list (if it is > 1). Computation induction proves

$$|merge\_adj\ xs| = (|xs| + 1)\ div\ 2 \tag{2.16}$$

2.5.1 Functional Correctness

The third and the last property prove functional correctness of *msort_bu*:

$$\begin{aligned}
&\textstyle\bigcup\#\ (image\_mset\ mset\ (mset\ (merge\_adj\ xss)))\\
&= \textstyle\bigcup\#\ (image\_mset\ mset\ (mset\ xss))\\
&mset\ (merge\_all\ xss) = \textstyle\bigcup\#\ (mset\ (map\ mset\ xss)) \qquad (2.17)\\
&mset\ (msort\_bu\ xs) = mset\ xs
\end{aligned}$$

$$\begin{aligned}
&(\forall xs \in set\ xss.\ sorted\ xs) \longrightarrow (\forall xs \in set\ (merge\_adj\ xss).\ sorted\ xs)\\
&(\forall xs \in set\ xss.\ sorted\ xs) \longrightarrow sorted\ (merge\_all\ xss) \qquad (2.18)\\
&sorted\ (msort\_bu\ xs)
\end{aligned}$$

where $\bigcup\#$ is the union of a multiset of multisets and *image_mset* :: ('a ⇒ 'b) ⇒ 'a *multiset* ⇒ 'b *multiset* applies a function to all elements of a multiset. The proof of each proposition may use the preceding proposition and the propositions (2.10) and (2.12). The propositions about *merge_adj* and *merge_all* are proved by computation inductions.

2.5.2 Running Time Analysis

Again, we count only comparisons:

$c\_merge\_adj :: {'a}\ list\ list \Rightarrow nat$

$c\_merge\_adj\ [] = 0$
$c\_merge\_adj\ [\_] = 0$
$c\_merge\_adj\ (xs\ \#\ ys\ \#\ zss) = c\_merge\ xs\ ys + c\_merge\_adj\ zss$

$c\_merge\_all :: {'a}\ list\ list \Rightarrow nat$

$c\_merge\_all\ [] = 0$
$c\_merge\_all\ [xs] = 0$
$c\_merge\_all\ xss = c\_merge\_adj\ xss + c\_merge\_all\ (merge\_adj\ xss)$

$c\_msort\_bu :: {'a}\ list \Rightarrow nat$

$c\_msort\_bu\ xs = c\_merge\_all\ (map\ (\lambda x.\ [x])\ xs)$

By simple computation inductions we obtain:

$$even\ |xss| \wedge (\forall xs{\in}set\ xss.\ |xs| = m) \longrightarrow$$
$$(\forall xs{\in}set\ (merge\_adj\ xss).\ |xs| = 2 \cdot m) \tag{2.19}$$
$$(\forall xs{\in}set\ xss.\ |xs| = m) \longrightarrow c\_merge\_adj\ xss \leqslant m \cdot |xss| \tag{2.20}$$

using (2.13) for (2.19) and (2.15) for (2.20).

Lemma 2.5. $(\forall xs{\in}set\ xss.\ |xs| = m) \wedge |xss| = 2^k \longrightarrow c\_merge\_all\ xss \leqslant m \cdot k \cdot 2^k$

Proof. The proof is by induction on the computation of *merge_all*. We concentrate on the nontrivial recursive case arising from the third equation. We assume $1 < |xss|$, $\forall xs{\in}set\ xss.\ |xs| = m$ and $|xss| = 2^k$. Clearly $k \geqslant 1$ and thus $even\ |xss|$. Thus (2.19) implies $\forall xs{\in}set\ (merge\_adj\ xss).\ |xs| = 2 \cdot m$. From $length\ xss > 1$ it follows that xss is of the form $xs\ \#\ ys\ \#\ xss_2$ and thus

$|merge\_adj\ xss| = |merge\ xs\ ys\ \#\ merge\_adj\ xss_2|$
$= 1 + |merge\_adj\ xss_2|$
$= 1 + (|xss_2| + 1)\ div\ 2$ using (2.16)
$= 2^{k - 1}$ using $|xss| = 2^k$ and $xss = xs\ \#\ ys\ \#\ xss_2$ by arithmetic

Let $yss = merge\_adj\ xss$. We can prove the lemma:

$c\_merge\_all\ xss = c\_merge\_adj\ xss + c\_merge\_all\ yss$
$\leqslant m \cdot 2^k + c\_merge\_all\ yss$ using $|xss| = 2^k$ and (2.20)
$\leqslant m \cdot 2^k + 2 \cdot m \cdot (k - 1) \cdot 2^{k - 1}$
by IH using $\forall xs{\in}set\ yss.\ |xs| = 2 \cdot m$ and $|yss| = 2^{k - 1}$
$= m \cdot k \cdot 2^k$ □

Setting $m = 1$ we obtain the same upper bound as for top-down merge sort in Lemma 2.4:

Corollary 2.6. $|xs| = 2^k \longrightarrow c\_msort\_bu\ xs \leqslant k \cdot 2^k$

2.6 Efficient Bottom-Up Merge Sort [2]

One disadvantage of all the sorting functions we have seen so far, is that even in the best case they do not improve upon the $n * lg\ n$ bound. For example, given the sorted input [1, 2, 3, 4, 5], *msort_bu* will as a first step create [[1], [2], [3], [4], [5]] and then merge this list of lists recursively.

A slight variation of bottom-up merge sort, sometimes referred to as "natural merge sort," first partitions the input into its constituent ascending and descending subsequences and only then starts merging. In the above example we would get *merge_all* [[1, 2, 3, 4, 5]] which returns immediately with the result [1, 2, 3, 4, 5]. Assuming that creating ascending and descending subsequences is of linear complexity this yields a best-case performance that is linear in the number of list elements.

The function *sequences*, which is defined mutually recursively with *asc* and *desc* computes the initial list of lists:

```
sequences :: 'a list ⇒ 'a list list
sequences (a # b # xs)
= (if b < a then desc b [a] xs else asc b ((#) a) xs)
sequences [x] = [[x]]
sequences [] = []

asc :: 'a ⇒ ('a list ⇒ 'a list) ⇒ 'a list ⇒ 'a list list
asc a as (b # bs)
= (if ¬ b < a then asc b (as ∘ (#) a) bs
   else as [a] # sequences (b # bs))
asc a as [] = [as [a]]

desc :: 'a ⇒ 'a list ⇒ 'a list ⇒ 'a list list
desc a as (b # bs)
= (if b < a then desc b (a # as) bs
   else (a # as) # sequences (b # bs))
desc a as [] = [a # as]
```

[2] AFP/Efficient-Mergesort/Efficient_Sort.thy

Natural merge sort is just a combination of *merge_all* and *sequences*

```
nmsort :: 'a list ⇒ 'a list
nmsort xs = merge_all (sequences xs)
```

2.6.1 Functional Correctness

We have

$$\bigcup\# \; (\mathit{image\_mset}\ \mathit{mset}\ (\mathit{mset}\ (\mathit{sequences}\ xs))) = \mathit{mset}\ xs \qquad (2.21)$$
$$\mathit{mset}\ (\mathit{nmsort}\ xs) = \mathit{mset}\ xs \qquad (2.22)$$

where we use (2.21) together with (2.17) in order to show (2.22). Moreover, we have

$$\forall\, x{\in}\mathit{set}\ (\mathit{sequences}\ xs).\ \mathit{sorted}\ x \qquad (2.23)$$
$$\mathit{sorted}\ (\mathit{nmsort}\ xs) \qquad (2.24)$$

where we use (2.23) together with (2.18) to obtain (2.24).

2.6.2 Running Time Analysis

Once more, we only count comparisons:

```
c_sequences :: 'a list ⇒ nat
c_sequences (a # b # xs)
= 1 + (if b < a then c_desc b xs else c_asc b xs)
c_sequences [] = 0
c_sequences [_] = 0

c_asc :: 'a ⇒ 'a list ⇒ nat
c_asc a (b # bs)
= 1 + (if ¬ b < a then c_asc b bs else c_sequences (b # bs))
c_asc _ [] = 0

c_desc :: 'a ⇒ 'a list ⇒ nat
c_desc a (b # bs)
= 1 + (if b < a then c_desc b bs else c_sequences (b # bs))
c_desc _ [] = 0

c_nmsort :: 'a list ⇒ nat
c_nmsort xs = c_sequences xs + c_merge_all (sequences xs)
```

Before talking about *c_nmsort*, we need a variant of Lemma 2.5 that also works for lists whose length is not a power of two (since the result of *sequences* will usually not satisfy this property).

To this end, we will need the following two results, which we prove by two simple computation inductions together with (2.15) and (2.13):

$$c\_merge\_adj\ xss \leqslant |concat\ xss| \tag{2.25}$$
$$|concat\ (merge\_adj\ xss)| = |concat\ xss| \tag{2.26}$$

Lemma 2.7. $c\_merge\_all\ xss \leqslant |concat\ xss| \cdot \lceil \lg\ |xss| \rceil$

Proof. The proof is by induction on the computation of *c_merge_all*. We concentrate on the nontrivial recursive case arising from the third equation. It follows that *xss* is of the form $xs \mathbin{\#} ys \mathbin{\#} zss$. Further note that

$$\lceil \lg\ (n + 2) \rceil = \lceil \lg\ ((n + 1)\ div\ 2 + 1) \rceil + 1 \tag{$\star$}$$

for arbitrary n.

Now, let $m = |concat\ xss|$. Then we have

$$\begin{aligned}
&c\_merge\_all\ xss = c\_merge\_adj\ xss + c\_merge\_all\ (merge\_adj\ xss) \\
&\leqslant m + c\_merge\_all\ (merge\_adj\ xss) && \text{using (2.25)} \\
&\leqslant m + |concat\ (merge\_adj\ xss)| \cdot \lceil \lg\ |merge\_adj\ xss| \rceil && \text{by IH} \\
&= m + m \cdot \lceil \lg\ |merge\_adj\ xss| \rceil && \text{by (2.26)} \\
&= m + m \cdot \lceil \lg\ ((|xss| + 1)\ div\ 2) \rceil && \text{by (2.16)} \\
&= m + m \cdot \lceil \lg\ ((|zss| + 1)\ div\ 2 + 1) \rceil \\
&= m \cdot (\lceil \lg\ ((|zss| + 1)\ div\ 2 + 1) \rceil + 1) \\
&= m \cdot \lceil \lg\ (|zss| + 2) \rceil && \text{by } (\star) \\
&= m \cdot \lceil \lg\ |xss| \rceil && \square
\end{aligned}$$

Two simple computation inductions yield:

$$|sequences\ xs| \leqslant |xs| \tag{2.27}$$
$$c\_sequences\ xs \leqslant |xs| - 1 \tag{2.28}$$

At this point we obtain an upper bound on the number of comparisons required by *c_nmsort*.

Lemma 2.8. $|xs| = n \longrightarrow c\_nmsort\ xs \leqslant n + n \cdot \lceil \lg\ n \rceil$

Proof. Note that

$$c\_merge\_all\ (sequences\ xs) \leqslant n \cdot \lceil \lg\ n \rceil \tag{$\star$}$$

as shown by the derivation:

$$\begin{aligned}
&c\_merge\_all\ (sequences\ xs) \\
&\leqslant n \cdot \lceil \lg\ |sequences\ xs| \rceil && \text{by Lemma 2.7 with } xss = sequences\ xs \\
&\leqslant n \cdot \lceil \lg\ n \rceil && \text{by (2.27)}
\end{aligned}$$

We conclude the proof by:

$$\begin{aligned} & c\_nmsort\ xs = c\_sequences\ xs + c\_merge\_all\ (sequences\ xs) \\ & \leqslant n + n \cdot \lceil \lg n \rceil && \text{using (2.28) and } (\star) \quad \square \end{aligned}$$

2.7 Stability

A sorting function is called **stable** if the order of equal elements is preserved. However, this only makes a difference if elements are not identified with their keys, as we have done. Let us assume instead that sorting is parameterized with a key function $f :: 'a \Rightarrow 'k$ that maps an element to its key and that the keys $'k$ are linearly ordered, not the elements. The specification of a sorting function *sort_key* must fulfil is now

$$\begin{aligned} & mset\ (sort\_key\ f\ xs) = mset\ xs \\ & sorted\ (map\ f\ (sort\_key\ f\ xs)) \end{aligned}$$

Assuming (for simplicity) we are sorting pairs of keys and some attached information, stability means that sorting $[(2, x), (1, z), (1, y)]$ yields $[(1, z), (1, y), (2, x)]$ and not $[(1, y), (1, z), (2, x)]$. That is, if we extract all elements with the same key *after* sorting *xs* they should be in the same order as in *xs*:

$$filter\ (\lambda y.\ f\ y = k)\ (sort\_key\ f\ xs) = filter\ (\lambda y.\ f\ y = k)\ xs$$

We will now define and examine insertion sort adapted to keys:

$$\begin{aligned} & insort\_key :: ('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'b\ list \Rightarrow 'b\ list \\ & insort\_key\ f\ x\ [] = [x] \\ & insort\_key\ f\ x\ (y \# ys) \\ & = (if\ f\ x \leqslant f\ y\ then\ x \# y \# ys\ else\ y \# insort\_key\ f\ x\ ys) \\ \\ & isort\_key :: ('a \Rightarrow 'k) \Rightarrow 'a\ list \Rightarrow 'a\ list \\ & isort\_key\ \_\ [] = [] \\ & isort\_key\ f\ (x \# xs) = insort\_key\ f\ x\ (isort\_key\ f\ xs) \end{aligned}$$

The proofs of the functional correctness properties

$$\begin{aligned} & mset\ (isort\_key\ f\ xs) = mset\ xs \\ & sorted\ (map\ f\ (isort\_key\ f\ xs)) && (2.29) \end{aligned}$$

are completely analogous to their counterparts for plain *isort*.

The proof of stability uses three auxiliary properties:

$$(\forall x \in set\ xs.\ f\ a \leqslant f\ x) \longrightarrow insort\_key\ f\ a\ xs = a\ \#\ xs \tag{2.30}$$

$$\neg\ P\ x \longrightarrow filter\ P\ (insort\_key\ f\ x\ xs) = filter\ P\ xs \tag{2.31}$$

$$\begin{aligned}&sorted\ (map\ f\ xs) \wedge P\ x \longrightarrow\\ &filter\ P\ (insort\_key\ f\ x\ xs) = insort\_key\ f\ x\ (filter\ P\ xs)\end{aligned} \tag{2.32}$$

The first one is proved by a case analysis on *xs*. The other two are proved by induction on *xs*, using (2.30) in the proof of (2.32).

Lemma 2.9 (stability of *isort_key*).
$filter\ (\lambda y.\ f\ y = k)\ (isort\_key\ f\ xs) = filter\ (\lambda y.\ f\ y = k)\ xs$

Proof. The proof is by induction on *xs*. The base cases is trivial. In the induction step we consider the list $a\ \#\ xs$ and perform a case analysis. If $f\ a \neq k$ the claim follows by IH using (2.31). Now assume $f\ a = k$:

$$\begin{aligned}
&filter\ (\lambda y.\ f\ y = k)\ (isort\_key\ f\ (a\ \#\ xs))\\
&= filter\ (\lambda y.\ f\ y = k)\ (insort\_key\ f\ a\ (isort\_key\ f\ xs))\\
&= insort\_key\ f\ a\ (filter\ (\lambda y.\ f\ y = k)\ (isort\_key\ f\ xs))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{using } f\ a = k\text{, (2.32), (2.29)}\\
&= insort\_key\ f\ a\ (filter\ (\lambda y.\ f\ y = k)\ xs) \qquad \text{by IH}\\
&= a\ \#\ filter\ (\lambda y.\ f\ y = k)\ xs \qquad \text{using } f\ a = k \text{ and (2.30)}\\
&= filter\ (\lambda y.\ f\ y = k)\ (a\ \#\ xs) \qquad \text{using } f\ a = k
\end{aligned}$$
□

Exercises

Exercise 2.3. Show that the result of *sequences xs* does not contain empty lists:

theorem "$\forall x \in set\ (sequences\ xs).\ x \neq []$"

3

Selection (Manuel Eberl)

Part II

Search Trees

Binary Trees [1]

Binary trees are defined as a recursive data type:

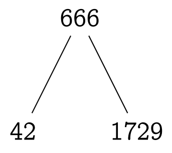
datatype $'a\ tree = Leaf \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

The following syntactic sugar is sprinkled on top:

$$\begin{aligned} \langle\rangle &\equiv Leaf \\ \langle l,\ x,\ r\rangle &\equiv Node\ l\ x\ r \end{aligned}$$

Because most of our tress will be binary trees, we drop the "binary" most of the time and have also called the type merely *tree*.

When displaying a tree in the usual graphical manner we show only the *Node*s. For example, $\langle\langle\langle\rangle,\ 42,\ \langle\rangle\rangle,\ 666,\ \langle\langle\rangle,\ 1729,\ \langle\rangle\rangle\rangle$ is displayed like this:



The (label of the) **root** node is 666. The **depth** (or **level**) of some node (or leaf) in a tree is the distance from the root. We use these concepts only informally.

4.1 Basic Functions

Two canonical functions on data types are *set* and *map*:

[1] isabelle/src/HOL/Library/Tree.thy

$set\_tree :: {'a}\ tree \Rightarrow {'a}\ set$

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

$map\_tree :: ({'a} \Rightarrow {'b}) \Rightarrow {'a}\ tree \Rightarrow {'b}\ tree$

$map\_tree\ f\ \langle\rangle = \langle\rangle$
$map\_tree\ f\ \langle l, x, r\rangle = \langle map\_tree\ f\ l, f\ x, map\_tree\ f\ r\rangle$

The *inorder*, *preorder* and *postorder* traversals (we omit the latter) list the elements in a tree in a particular order:

$inorder :: {'a}\ tree \Rightarrow {'a}\ list$

$inorder\ \langle\rangle = []$
$inorder\ \langle l, x, r\rangle = inorder\ l\ @\ [x]\ @\ inorder\ r$

$preorder :: {'a}\ tree \Rightarrow {'a}\ list$

$preorder\ \langle\rangle = []$
$preorder\ \langle l, x, r\rangle = x\ \#\ preorder\ l\ @\ preorder\ r$

The two size functions count the number of *Node*s and *Leaf*s in a tree:

$size :: {'a}\ tree \Rightarrow nat$

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

$size1 :: {'a}\ tree \Rightarrow nat$

$|\langle\rangle|_1 = 1$
$|\langle l, \_, r\rangle|_1 = |l|_1 + |r|_1$

The syntactic sugar $|t|$ for *size* t and $|t|_1$ for *size*1 t is only used in this text, not in the Isabelle theories.

Induction proves a convenient fact that explains the name *size*1:

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

The height and the minimal height of a tree are defined as follows:

$height :: 'a\ tree \Rightarrow nat$

$h\ \langle\rangle = 0$
$h\ \langle l,\ \_,\ r\rangle = max\ (h\ l)\ (h\ r) + 1$

$min\_height :: 'a\ tree \Rightarrow nat$

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

You can think of them as the longest and shortest (cycle-free) path from the root to a leaf.

! The real names of these functions are *height* and *min_height*. The abbreviations *h* and *mh* are only used in this text, not in the Isabelle theories.

The obvious properties $h\ t \leqslant |t|$ and $mh\ t \leqslant h\ t$ and the following classical properties have easy inductive proofs:

$$2^{mh\ t} \leqslant |t|_1 \qquad |t|_1 \leqslant 2^{h\ t}$$

We will simply use these fundamental properties without referring to them by a name or number.

Exercise 4.1. Function *inorder* has quadratic complexity because the running time of (@) is linear in the length of its first argument. Define a function $inorder2 :: 'a\ tree \Rightarrow 'a\ list \Rightarrow 'a\ list$ that avoids (@) but accumulates the result in its second parameter via (#) only. Its running should be linear in the size of the tree. Prove $inorder2\ t\ xs = inorder\ t\ @\ xs$.

4.2 Complete Trees

A **complete tree** is one where all the leaves are on the same level. An example is shown in Figure 4.1.

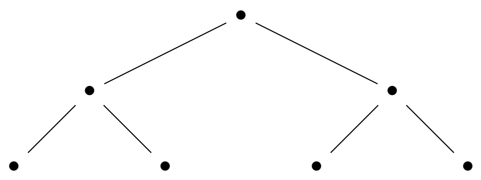


Fig. 4.1. A complete tree

We define the predicate *complete* recursively:

$$\begin{aligned}&\mathit{complete} :: {}'a\ \mathit{tree} \Rightarrow \mathit{bool}\\ &\mathit{complete}\ \langle\rangle = \mathit{True}\\ &\mathit{complete}\ \langle l,\ \_,\ r\rangle = (\mathit{complete}\ l \wedge \mathit{complete}\ r \wedge h\ l = h\ r)\end{aligned}$$

This recursive definition is equivalent with the above definition that all leaves must have the same distance from the root. Formally:

Lemma 4.1. $\mathit{complete}\ t \longleftrightarrow mh\ t = h\ t$

Proof. The proof is by induction and case analyses on *min* and *max*. □

The classic property of complete trees is that their number of leaves is a power of 2. The proof is a simple induction:

Lemma 4.2. $\mathit{complete}\ t \longrightarrow |t|_1 = 2^{h\ t}$

Next we prove that in incomplete trees, the height is strictly greater than the logarithm of the number of leaves. That is, we prove that the $\leqslant$ in $2^{mh\ t} \leqslant |t|_1$ and $|t|_1 \leqslant 2^{h\ t}$ becomes $<$ for incomplete trees:

Lemma 4.3. $\neg\ \mathit{complete}\ t \longrightarrow |t|_1 < 2^{h\ t}$

Proof. The proof is by induction. We focus in the induction step where $t = \langle l,\ x,\ r\rangle$. If t is incomplete, there are a number of cases and we prove $|t|_1 < 2^{h\ t}$ in each case. If $h\ l \neq h\ r$, consider the case $h\ l < h\ r$ (the case $h\ r < h\ l$ is symmetric). From $2^{h\ l} < 2^{h\ r}$, $|l|_1 \leqslant 2^{h\ l}$ and $|r|_1 \leqslant 2^{h\ r}$ the claim follows: $|t|_1 = |l|_1 + |r|_1 \leqslant 2^{h\ l} + 2^{h\ r} < 2 \cdot 2^{h\ r} = 2^{h\ t}$. If $h\ l = h\ r$, then either l or r must be incomplete. We consider the case $\neg\ \mathit{complete}\ l$ (the case $\neg\ \mathit{complete}\ r$ is symmetric). From the IH $|l|_1 < 2^{h\ l}$, $|r|_1 \leqslant 2^{h\ r}$ and $h\ l = h\ r$ the claim follows: $|t|_1 = |l|_1 + |r|_1 < 2^{h\ l} + 2^{h\ r} = 2 \cdot 2^{h\ r} = 2^{h\ t}$. □

Lemma 4.4. $\neg\ \mathit{complete}\ t \longrightarrow 2^{mh\ t} < |t|_1$

The proof of this lemma is completely analogous to the previous proof except that one also needs to use Lemma 4.1.

From the contrapositive of Lemma 4.3 one obtains $|t|_1 = 2^{h\ t} \longrightarrow \mathit{complete}\ t$, the converse of Lemma 4.2. Thus we arrive at:

Corollary 4.5. $\mathit{complete}\ t \longleftrightarrow |t|_1 = 2^{h\ t}$

The complete trees are precisely the ones where the height is exactly the logarithm of the number of leaves.

Exercise 4.2. Consult the predefined function *subtrees* (in theory *Tree*) for a precise definition of what a subtree is. Define a function *mcs* that computes a maximal complete subtree of some given tree. You are allowed only one traversal of the input but you may freely compute the height of trees and may even compare trees for equality. In particular you are not allowed to use *complete* or *subtrees*.

Prove that *mcs* returns a complete subtree and that it is maximal (in height):

$$u \in subtrees\ t \wedge complete\ u \longrightarrow h\ u \leqslant h\ (mcs\ t)$$

Bonus: get rid of any tree equality tests in *mcs*.

4.3 Balanced Trees

A balanced tree is one where the leaves may occur not just at the lowest level but also one level above:

$balanced :: {'a}\ tree \Rightarrow bool$

$balanced\ t = (h\ t - mh\ t \leqslant 1)$

An example of a balanced tree is shown in Figure 4.2. You can think of a balanced tree as a complete tree with (possibly) some additional nodes one level below the last full level.

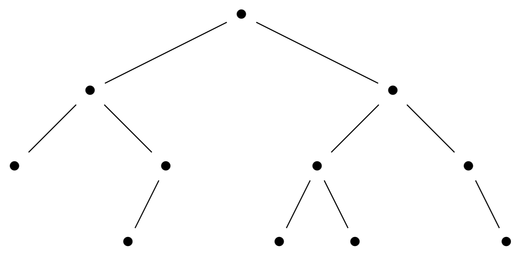


Fig. 4.2. A balanced tree

Balanced trees are important because among all the trees with the same number of nodes they have minimal height:

Lemma 4.6. $balanced\ s \wedge |s| \leqslant |t| \longrightarrow h\ s \leqslant h\ t$

Proof. The proof is by cases. If *complete s* then, by Lemma 4.2, $2^{h\ s} = |s|_1 \leqslant |t|_1 \leqslant 2^{h\ t}$ and thus $h\ s \leqslant h\ t$. Now assume $\neg$ *complete s*. Then Lemma 4.4

yields $2^{mh\ s} < |s|_1 \leqslant |t|_1 \leqslant 2^{h\ t}$ and thus $mh\ s < h\ t$. Furthermore we have $h\ s - mh\ s \leqslant 1$ (from *balanced s*), $h\ s \neq mh\ s$ (from Lemma 4.1) and $mh\ s \leqslant h\ s$, which together imply $mh\ s + 1 = h\ s$. With $mh\ s < h\ t$ this implies $h\ s \leqslant h\ t$. □

This is relevant for search trees because their height determines the worst case running time. Balanced trees are optimal in that sense.

The following lemma yields an explicit formula for the height of a balanced tree:

Lemma 4.7. $balanced\ t \longrightarrow h\ t = \lceil \lg |t|_1 \rceil$

Proof. The proof is by cases. If t is complete it follows from Lemma 4.2. Now assume t is incomplete. Then $h\ t = mh\ t + 1$ because *balanced t*, $mh\ t \leqslant h\ t$ and $complete\ t \longleftrightarrow mh\ t = h\ t$ (Lemma 4.1). Together with $|t|_1 \leqslant 2^{h\ t}$ this yields $|t|_1 \leqslant 2^{mh\ t\ +\ 1}$ and thus $\lg |t|_1 \leqslant mh\ t + 1$. By Lemma 4.4 we obtain $mh\ t < \lg |t|_1$. These two bounds for $\lg |t|_1$ together imply the claimed $h\ t = \lceil \lg |t|_1 \rceil$. □

In the same manner we also obtain:

Lemma 4.8. $balanced\ t \longrightarrow mh\ t = \lfloor \lg |t|_1 \rfloor$

4.3.1 Converting a List into a Balanced Tree

We will now see how to convert a list xs into a balanced tree t such that $inorder\ t = xs$. If the list is sorted, the result is a balanced binary search tree. The basic idea is to cut the list in two halves, turn them into balanced trees recursively and combine them. But cutting up the list in two halves explicitly would lead to an $n \cdot \lg\ n$ algorithm but we want a linear one. Therefore we use an additional *nat* parameter to tell us how much of the input list should be turned into a tree. The remaining list is returned with the tree:

```
bal :: nat ⇒ 'a list ⇒ 'a tree × 'a list
bal n xs
= (if n = 0 then (⟨⟩, xs)
   else let m = n div 2;
            (l, ys) = bal m xs;
            (r, zs) = bal (n − 1 − m) (tl ys)
        in (⟨l, hd ys, r⟩, zs))
```

The trick is not to chop xs in half but n because we assume that arithmetic is constant time. Hence bal runs in linear time (see Exercise 4.4).

Balancing all or some prefix of a list or tree is easily derived:

$bal\_list :: nat \Rightarrow \ 'a\ list \Rightarrow \ 'a\ tree$

$bal\_list\ n\ xs = fst\ (bal\ n\ xs)$

$balance\_list :: \ 'a\ list \Rightarrow \ 'a\ tree$

$balance\_list\ xs = bal\_list\ |xs|\ xs$

$bal\_tree :: nat \Rightarrow \ 'a\ tree \Rightarrow \ 'a\ tree$

$bal\_tree\ n\ t = bal\_list\ n\ (inorder\ t)$

$balance\_tree :: \ 'a\ tree \Rightarrow \ 'a\ tree$

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

Correctness

The following lemma clearly expresses that $bal\ n\ xs$ turns the prefix of length n of xs into a tree and returns the corresponding suffix of xs:

Lemma 4.9.
$n \leqslant |xs| \wedge bal\ n\ xs = (t,\ zs) \longrightarrow xs = inorder\ t\ @\ zs \wedge |t| = n$

Proof. The proof is by computation induction. The base case $n = 0$ is trivial. Now assume $n \neq 0$ and let $m = n\ div\ 2$ and $m' = n - 1 - m$. From $bal\ n\ xs = (t,\ zs)$ we obtain l, r and ys such that $bal\ m\ xs = (l,\ ys)$, $bal\ m'\ (tl\ ys) = (r,\ zs)$ and $t = \langle l,\ hd\ ys,\ r\rangle$. Because $m \leqslant |xs|$, the first induction hypothesis simplifies to $xs = inorder\ l\ @\ ys \wedge |l| = m$. This in turn implies $m' \leqslant |tl\ ys|$ and thus the second induction hypothesis simplifies to $tl\ ys = inorder\ r\ @\ zs \wedge |r| = m'$. The simplified induction hypotheses together with $t = \langle l,\ hd\ ys,\ r\rangle$ imply the claim $xs = inorder\ t\ @\ zs \wedge |t| = n$ because $ys \neq []$. □

The corresponding correctness properties of the derived functions are easy consequences:

$$\begin{aligned} n \leqslant |xs| &\longrightarrow inorder\ (bal\_list\ n\ xs) = take\ n\ xs \\ & inorder\ (balance\_list\ xs) = xs \\ n \leqslant |t| &\longrightarrow inorder\ (bal\_tree\ n\ t) = take\ n\ (inorder\ t) \\ & inorder\ (balance\_tree\ t) = inorder\ t \end{aligned}$$

To prove that *bal* returns a balanced tree we determine its height and minimal height.

Lemma 4.10. $n \leqslant |xs| \wedge bal\ n\ xs = (t,\ zs) \longrightarrow h\ t = \lceil \lg\ (n + 1) \rceil$

Proof. The proof structure is the same as for Lemma 4.9 and we reuse the variable names introduced there. In the induction step we obtain the simplified induction hypothesese $h\ l = \lceil \lg\ (m + 1) \rceil$ and $h\ r = \lceil \lg\ (m' + 1) \rceil$. This leads to

$$\begin{aligned} & h\ t = max\ (h\ l)\ (h\ r) + 1 \\ & = h\ l + 1 && \text{because } m' \leqslant m \\ & = \lceil \lg\ (m + 1) \rceil + 1 \\ & = \lceil \lg\ (n + 1) \rceil \end{aligned}$$

The last step employs this basic arithmetic lemma that holds for all $n :: nat$:

$$2 \leqslant n \longrightarrow \lceil \lg\ n \rceil = \lceil \lg\ ((n - 1)\ div\ 2 + 1) \rceil + 1 \qquad \square$$

The following complementary lemma is proved in the same way:

Lemma 4.11. $n \leqslant |xs| \wedge bal\ n\ xs = (t,\ zs) \longrightarrow mh\ t = \lfloor \lg\ (n + 1) \rfloor$

By definition of *balanced* and because $\lceil x \rceil - \lfloor x \rfloor \leqslant 1$ we obtain that *bal* (and consequently the functions that build on it) returns a balanced tree:

Corollary 4.12. $n \leqslant |xs| \wedge bal\ n\ xs = (t,\ ys) \longrightarrow balanced\ t$

4.3.2 Exercises

Exercise 4.3. Find a formula B such that $balanced\ \langle l,\ x,\ r \rangle = B$ where B may only contain the functions *balanced*, *complete*, *height*, arithmetic and boolean operations, l and r, but in particular not *min_height* or *Node*. Prove $balanced\ \langle l,\ x,\ r \rangle = B$.

Exercise 4.4. Prove that the running time of function *bal* is linear in its first argument. (Hint: you need to refer to *bal* as *Balance.bal*.)

4.4 Augmented Trees [2]

A tree of type *'a tree* only stores elements of type *'a*. However, it is frequently necessary to store some additional information of type *'b* in each node too, usually for efficiency reasons. Typical examples are:

[2] isabelle/src/HOL/Data_Structures/Tree2.thy

- The size or the height of the tree, the sum of elements in the tree. Because recomputing them requires traversing the whole tree.
- Lookup tables where each key of type $'a$ is associated with a value of type $'b$.

In this case we simply work with trees of type $('a \times 'b)$ *tree* and call them **augmented trees**. As a result we need to redefine a few functions that should ignore the additional information. For example, function *inorder*, when applied to an augmented tree, should have return type $'a$ *list*. Thus we redefine it in the obvious way:

```
inorder :: ('a × 'b) tree ⇒ 'a list
inorder ⟨⟩ = []
inorder ⟨l, (a, _), r⟩ = inorder l @ a # inorder r
```

Another example is *set_tree* :: $('a \times 'b)$ *tree* $\Rightarrow$ $'a$ *set*. In general, if a function f is originally defined on type $'a$ *tree* but should ignore the $'b$-values in an $('a \times 'b)$ *tree* then we assume that there is a corresponding revised definition of f on augmented trees that focuses on the $'a$-values just like *inorder* above does. Of course functions that do not depend on the information in the nodes, e.g. size and height, stay unchanged.

Note that there are two alternative redefinitions of *inorder* (and similar functions): *map fst* $\circ$ *Tree.inorder* or *Tree.inorder* $\circ$ *map_tree fst* where *Tree.inorder* is the original function.

4.4.1 Maintaining Augmented Trees

Maintaining the $'b$-values in a tree can be hidden inside a suitable smart version of *Node* that has only a constant time overhead. Take the example of augmentation by size:

```
sz :: ('a × nat) tree ⇒ nat
sz ⟨⟩ = 0
sz ⟨_, (_, n), _⟩ = n

node_sz :: ('a × nat) tree ⇒ 'a ⇒ ('a × nat) tree ⇒ ('a × nat) tree
node_sz l a r = ⟨l, (a, sz l + sz r + 1), r⟩
```

A $('a \times nat)$ *tree* satisfies *invar_sz* if the size annotation of every node is computed from the subtrees as specified in *node_sz*:

$invar\_sz :: ('a \times nat)\ tree \Rightarrow bool$

$invar\_sz\ \langle\rangle = True$
$invar\_sz\ \langle l,\ (\_,\ n),\ r\rangle$
$= (invar\_sz\ l \wedge invar\_sz\ r \wedge n = sz\ l + sz\ r + 1)$

This property is preserved by $node\_sz$ $(invar\_sz\ l \wedge invar\_sz\ r \longrightarrow invar\_sz\ (node\_sz\ l\ a\ r))$ and it guarantees that sz returns the size:

$$invar\_sz\ t \longrightarrow sz\ t = |t|$$

We can generalize this example easily. Assume we have a function $f :: 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$ and a constant $zero :: 'b$ which we iterate over the tree:

$F :: ('a \times 'b)\ tree \Rightarrow 'b$

$F\ \langle\rangle = zero$
$F\ \langle l,\ (a,\ \_),\ r\rangle = f\ (F\ l)\ a\ (F\ r)$

This generalizes the definition of size. Let $node\_f$ compute the $'b$-value from the $'b$-value of its children via f:

$b\_val :: ('a \times 'b)\ tree \Rightarrow 'b$

$b\_val\ \langle\rangle = zero$
$b\_val\ \langle\_,\ (\_,\ b),\ \_\rangle = b$

$node\_f :: ('a \times 'b)\ tree \Rightarrow 'a \Rightarrow ('a \times 'b)\ tree \Rightarrow ('a \times 'b)\ tree$

$node\_f\ l\ a\ r = \langle l,\ (a,\ f\ (b\_val\ l)\ a\ (b\_val\ r)),\ r\rangle$

If all $'b$-values are computed as in $node\_f$

$invar\_f :: ('a \times 'b)\ tree \Rightarrow bool$

$invar\_f\ \langle\rangle = True$
$invar\_f\ \langle l,\ (a,\ b),\ r\rangle$
$= (invar\_f\ l \wedge invar\_f\ r \wedge b = f\ (b\_val\ l)\ a\ (b\_val\ r))$

then all $'b$-values equal F: $invar\_f\ t \longrightarrow b\_val\ t = F\ t$.

4.4.2 Exercises

Exercise 4.5. Augment trees by a pair of a boolean and something else where the boolean indicates whether the tree is complete or not. Define *ch*, *node_ch* and *invar_ch* as in Section 4.4.1 and prove the following properties:

$$\mathit{invar\_ch}\ t \longrightarrow \mathit{ch}\ t = (\mathit{complete}\ t,\ ?)$$
$$\mathit{invar\_ch}\ l \wedge \mathit{invar\_ch}\ r \longrightarrow \mathit{invar\_ch}\ (\mathit{node\_ch}\ l\ a\ r)$$

Exercise 4.6. Assume type *'a* is of class *linorder* and augment each *Node* with the maximum value in that tree. Following Section 4.4.1 (but mind the *option* type!) define $\mathit{mx} :: ('a \times 'b)\ \mathit{tree} \Rightarrow 'b\ \mathit{option}$, *node_mx* and *invar_mx* and prove $\mathit{invar\_mx}\ t \longrightarrow \mathit{mx}\ t = (\mathit{if}\ t = \langle\rangle\ \mathit{then}\ \mathit{None}\ \mathit{else}\ \mathit{Some}\ (\mathit{Max}\ (\mathit{set\_tree}\ t)))$ where *Max* is the predefined maximum operator on finite, non-empty sets.

5

Binary Search Trees [1]

The purpose of this chapter is threefold: to introduce **binary search trees** (BSTs), to discuss their correctness proofs, and to provide a first example of an abstract data type, a notion discussed in more detail in the next chapter.

Search trees are a means for storing and accessing collections of elements efficiently. In particular they can support sets and maps. We concentrate on sets. We have already seen function *set_tree* that maps a tree to the set of its elements. This is an example of an **abstraction function** that maps concrete data structures to the abstract values that they represent.

BSTs require a linear ordering on the elements in the tree. For each node, the elements in the left subtree are smaller than the root, the elements in the right subtree are bigger:

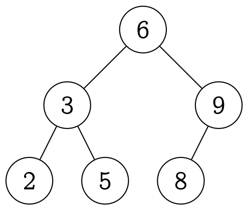
$bst :: ('a::linorder)\ tree \Rightarrow bool$

$bst\ \langle\rangle = True$

$bst\ \langle l,\ a,\ r\rangle$

$= (bst\ l \wedge bst\ r \wedge (\forall x{\in}set\_tree\ l.\ x < a) \wedge (\forall x{\in}set\_tree\ r.\ a < x))$

This is an example of a (for space reasons balanced) BST:



[1] isabelle/src/HOL/Data_Structures/Tree_Set.thy

It is obvious how to search for an element in a BST by comparing the element with the root and descending into one of the two subtrees if you have not found it yet. In the worst case this takes time proportional to the height of the tree. In later chapters we discuss a number of methods for ensuring that the height of the tree is logarithmic in its size. For now we ignore all efficiency considerations and permit our BSTs to degenerate.

5.1 Interfaces

Trees are concrete data types that provide the building blocks for realizing abstract data types like sets. The abstract type has a fixed interface, i.e. set of operations, through which the values of the abstract type can be manipulated. The interface hides all implementation detail. In the Search Tree part of the book we focus on the abstract type of sets with the following interface:

$empty :: {'s}$
$insert :: {'a} \Rightarrow {'s} \Rightarrow {'s}$
$delete :: {'a} \Rightarrow {'s} \Rightarrow {'s}$
$isin :: {'s} \Rightarrow {'a} \Rightarrow bool$

where $'s$ is the type of sets of elements of type $'a$. Most of our implementations of sets will be based on variants of BSTs and will require a linear order on $'a$, but the general interface does not require this. The correctness of an implementation of this interface will be proved by relating it back to HOL's type $'a\ set$ via an abstraction function, e.g. *set_tree*.

5.2 Implementing Sets via unbalanced BSTs

So far we have compared elements via $=$, $\leqslant$ and $<$. Now we switch to a comparator-based approach

datatype $cmp\_val = LT \mid EQ \mid GT$

$cmp :: ({'a}{::}\ linorder) \Rightarrow {'a} \Rightarrow cmp\_val$
$cmp\ x\ y = (\mathit{if}\ x < y\ \mathit{then}\ LT\ \mathit{else}\ \mathit{if}\ x = y\ \mathit{then}\ EQ\ \mathit{else}\ GT)$

We will frequently phrase algorithms in terms of *cmp*, *LT*, *EQ* and *GT* instead of $<$, $=$ and $>$. This leads to more symmetric code. If some type comes with its own primitive *cmp* function this can yield a speed-up over the above generic *cmp* function.

Below you find an implementations of the set interface in terms of BSTs. Functions *isin* and *insert* are self-explanatory. Deletion is more interesting.

```
empty = ⟨⟩

isin ⟨⟩ x = False
isin ⟨l, a, r⟩ x
= (case cmp x a of LT ⇒ isin l x | EQ ⇒ True | GT ⇒ isin r x)

insert x ⟨⟩ = ⟨⟨⟩, x, ⟨⟩⟩
insert x ⟨l, a, r⟩ = (case cmp x a of
                        LT ⇒ ⟨insert x l, a, r⟩ |
                        EQ ⇒ ⟨l, a, r⟩ |
                        GT ⇒ ⟨l, a, insert x r⟩)

delete x ⟨⟩ = ⟨⟩
delete x ⟨l, a, r⟩
= (case cmp x a of
    LT ⇒ ⟨delete x l, a, r⟩ |
    EQ ⇒ if r = ⟨⟩ then l
           else let (a', r') = split_min r in ⟨l, a', r'⟩ |
    GT ⇒ ⟨l, a, delete x r⟩)

split_min :: 'a tree ⇒ 'a × 'a tree
split_min ⟨l, a, r⟩
= (if l = ⟨⟩ then (a, r) else let (x, l') = split_min l in (x, ⟨l', a, r⟩))
```

5.2.1 Deletion

Function *delete* deletes a from $\langle l,\ a,\ r\rangle$ (where $r \neq \langle\rangle$) by replacing a with a' and r by r' where

a' is the leftmost (least) element of r, also called the inorder successor of a,
r' is the remainder of r after removing a'.

We call this **deletion by replacing**. Of course one can also obtain a' as the inorder predecessor of a in l.

An alternative is to delete a from $\langle l,\ a,\ r\rangle$ by "joining" l and r:

```
delete2 :: 'a ⇒ 'a tree ⇒ 'a tree
delete2 _ ⟨⟩ = ⟨⟩
delete2 x ⟨l, a, r⟩ = (case cmp x a of
                        LT ⇒ ⟨delete2 x l, a, r⟩ |
                        EQ ⇒ join l r |
                        GT ⇒ ⟨l, a, delete2 x r⟩)

join :: 'a tree ⇒ 'a tree ⇒ 'a tree
join t ⟨⟩ = t
join ⟨⟩ t = t
join ⟨t₁, a, t₂⟩ ⟨t₃, b, t₄⟩
= (case join t₂ t₃ of
    ⟨⟩ ⇒ ⟨t₁, a, ⟨⟨⟩, b, t₄⟩⟩ |
    ⟨u₂, x, u₃⟩ ⇒ ⟨⟨t₁, a, u₂⟩, x, ⟨u₃, b, t₄⟩⟩)
```

We call this **deletion by joining**. The characteristic property of *join* is *inorder* (*join l r*) = *inorder l* @ *inorder r*.

The definition of *join* may appear needlessly complicated. Why not this much simpler version:

```
join0 t ⟨⟩ = t
join0 ⟨⟩ t = t
join0 ⟨t₁, a, t₂⟩ ⟨t₃, b, t₄⟩ = ⟨t₁, a, ⟨join0 t₂ t₃, b, t₄⟩⟩
```

Because with this version of *join*, deletion may almost double the height of the tree, in contrast to *join* and also deletion by replacing, where the height cannot increase:

Exercise 5.1. First prove that *join* behaves well:

$h\ (join\ l\ r) \leqslant max\ (h\ l)\ (h\ r) + 1$

Now show that *join0* behaves badly: Find an upper bound *ub* of $h\ (join0\ l\ r)$ such that *ub* is a function of $h\ l$ and $h\ r$. Prove $h\ (join0\ l\ r) \leqslant ub$ and prove that *ub* is a tight upper bound for complete *l* and *r*.

We focus on *delete* in the rest of the chapter.

5.3 Correctness

Why is the above implementation correct? Roughly speaking, because the implementations of *empty*, *insert*, *delete* and *isin* on type *'a tree* simulate the behaviour of $\{\}$, $\cup$, $-$ and $\in$ on type *'a set*. Taking the abstraction function into account we can formulate the simulation precisely:

$$set\_tree\ empty = \{\}$$
$$set\_tree\ (insert\ x\ t) = set\_tree\ t \cup \{x\}$$
$$set\_tree\ (delete\ x\ t) = set\_tree\ t - \{x\}$$
$$isin\ t\ x = (x \in set\_tree\ t)$$

However, the implementation only works correctly on BSTs. Therefore we need to add the precondition *bst t* to all but the first property. But why are we permitted to assume this precondition? Only because *bst* is an **invariant** of this implementation: *bst* holds for *empty*, and both *insert* and *delete* preserve *bst*. Therefore every tree that can be manufactured through the interface is a BST. Of course this adds another set of proof obligations for correctness, **invariant preservation**:

$$bst\ empty$$
$$bst\ t \longrightarrow bst\ (insert\ x\ t)$$
$$bst\ t \longrightarrow bst\ (delete\ x\ t)$$

When looking at the abstract data type of sets from the user (or 'client') perspective, we would call the collection of all proof obligations for the correctness of an implementation the **specification** of the abstract type.

Exercise 5.2. Verify the implementation in Section 5.2 by showing all the proof obligations above, without the detour via sorted lists explained below.

5.4 Correctness Proofs

It turns out that direct proofs of the properties in previous section are cumbersome — at least for *delete* and for proof assistants like Isabelle. Yet the correctness of the implementation is quite obvious to most (functional) programmers. Which is why most algorithm texts do not spend any time on functional correctness of search trees and concentrate on non-obvious structural properties that imply the logarithmic height of the trees — of course our simple BSTs do not guarantee the latter.

We will now present how the vague notion of "obvious" can be concretized and automated to such a degree that we do not need to discuss functional correctness of search tree implementations again in this book. This is because our approach is quite generic: it works not only for the BSTs in this chapter but also for the more efficient variants discussed in later chapters. The remainder of this section can be skipped if one is not interested in proof automation.

5.4.1 The Idea

The key idea [23] is to express *bst* and *set_tree* via *inorder*:

$$bst\ t = sorted\ (inorder\ t) \quad \text{and} \quad set\_tree\ t = set\ (inorder\ t)$$

where

$sorted :: 'a\ list \Rightarrow bool$

$sorted\ [] = True$
$sorted\ [x] = True$
$sorted\ (x \# y \# zs) = (x < y \land sorted\ (y \# zs))$

Note that this is "sorted w.r.t. $(<)$" whereas in the chapter on sorting *sorted* was defined as "sorted w.r.t. $(\leqslant)$".

Instead of showing directly that BSTs implement sets, we show that they implement an intermediate specification based on lists (and later that the list-based specification implies the set-based one). We can assume that the lists are *sorted* because they are abstractions of BSTs. Insertion and deletion on sorted lists can be defined as follows:

$ins\_list :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$

$ins\_list\ x\ [] = [x]$
$ins\_list\ x\ (a \# xs)$
$= (if\ x < a\ then\ x \# a \# xs$
$\quad else\ if\ x = a\ then\ a \# xs\ else\ a \# ins\_list\ x\ xs)$

$del\_list :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$

$del\_list\ \_\ [] = []$
$del\_list\ x\ (a \# xs) = (if\ x = a\ then\ xs\ else\ a \# del\_list\ x\ xs)$

The abstraction function from trees to lists is function *inorder*. The specification in Figure 5.1 expresses that *empty*, *insert*, *delete* and *isin* implement $[]$, *ins_list*, *del_list* and $\lambda xs\ x.\ x \in set\ xs$. One nice aspect of this specifica-

$inorder\ empty = []$
$sorted\ (inorder\ t) \longrightarrow inorder\ (insert\ x\ t) = ins\_list\ x\ (inorder\ t)$
$sorted\ (inorder\ t) \longrightarrow inorder\ (delete\ x\ t) = del\_list\ x\ (inorder\ t)$
$bst\ t \longrightarrow isin\ t\ x = (x \in set\_tree\ t)$

Fig. 5.1. List-based Specification of BSTs

tion is that it does not require us to prove invariant preservation explicitly: it follows from the fact (proved below) that *ins_list* and *del_list* preserve *sorted*.

5.4.2 BSTs Implement Sorted Lists — A Framework

We present a library of lemmas that automate the functional correctness proofs for the BSTs in this chapter and the more efficient variants in later chapters. This library is motivated by general considerations concerning the shape of formulas that arise during verification.

As a motivating example we examine how to prove

$$\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{inorder}\ (\mathit{insert}\ x\ t) = \mathit{ins\_list}\ x\ (\mathit{inorder}\ t)$$

The proof is by induction on t and we consider the case $t = \langle l,\ a,\ r\rangle$ such that $x < a$. Ideally the proof looks like this:

$$\begin{aligned}&\mathit{inorder}\ (\mathit{insert}\ x\ t) = \mathit{inorder}\ (\mathit{insert}\ x\ l)\ @\ a\ \#\ \mathit{inorder}\ r\\ &= \mathit{ins\_list}\ x\ (\mathit{inorder}\ l)\ @\ a\ \#\ \mathit{inorder}\ r\\ &= \mathit{ins\_list}\ x\ (\mathit{inorder}\ l\ @\ a\ \#\ \mathit{inorder}\ r) = \mathit{ins\_list}\ x\ t\end{aligned}$$

The first and last step are by definition, the second step by induction hypothesis, and the third step by lemmas (5.1) and (5.5) in Figure 5.2: the first lemma rewrites the assumption $\mathit{sorted}\ (\mathit{inorder}\ t)$ to $\mathit{sorted}\ (\mathit{inorder}\ l\ @\ [a]) \wedge \mathit{sorted}\ (a\ \#\ \mathit{inorder}\ r)$, thus allowing the second lemma to rewrite the term $\mathit{ins\_list}\ x\ (\mathit{inorder}\ l\ @\ a\ \#\ \mathit{inorder}\ r)$ to $\mathit{ins\_list}\ x\ (\mathit{inorder}\ l)\ @\ a\ \#\ \mathit{inorder}\ r$.

The lemma library in Figure 5.2 helps to prove the properties in Figure 5.1. These proofs will be by induction on t and lead to (possibly nested) tree constructor terms like $\langle\langle t_1,\ a_1,\ t_2\rangle,\ a_2,\ t_3\rangle$ where the t_i and a_i are variables. Evaluating *inorder* of such a tree leads to a list of the following form:

$$\mathit{inorder}\ t_1\ @\ a_1\ \#\ \mathit{inorder}\ t_2\ @\ a_2\ \#\ \ldots\ \#\ \mathit{inorder}\ t_n$$

Now we discuss the lemmas in Figure 5.2 that simplify the application of *sorted*, *ins_list* and *del_list* to such terms.

Terms of the form $\mathit{sorted}\ (\mathit{xs}_1\ @\ a_1\ \#\ \mathit{xs}_2\ @\ a_2\ \#\ \ldots\ \#\ \mathit{xs}_n)$ are decomposed into the following *basic* formulas

$$\begin{array}{ll}\mathit{sorted}\ (\mathit{xs}\ @\ [a]) & (\text{simulating } \forall x{\in}\mathit{set}\ \mathit{xs}.\ x < a)\\ \mathit{sorted}\ (a\ \#\ \mathit{xs}) & (\text{simulating } \forall x{\in}\mathit{set}\ \mathit{xs}.\ a < x)\\ a < b & \end{array}$$

by the rewrite rules (5.1)–(5.2). Lemmas (5.3)–(5.4) enable deductions from basic formulas.

Terms of the form $\mathit{ins\_list}\ x\ (\mathit{xs}_1\ @\ a_1\ \#\ \mathit{xs}_2\ @\ a_2\ \#\ \ldots\ \#\ \mathit{xs}_n)$ are rewritten with equation (5.5) (and the defining equations for *ins_list*) to

$$\mathit{sorted}\ (xs\ @\ y\ \#\ ys) = (\mathit{sorted}\ (xs\ @\ [y]) \wedge \mathit{sorted}\ (y\ \#\ ys)) \tag{5.1}$$

$$\begin{aligned}&\mathit{sorted}\ (x\ \#\ xs\ @\ y\ \#\ ys)\\ &= (\mathit{sorted}\ (x\ \#\ xs) \wedge x < y \wedge \mathit{sorted}\ (xs\ @\ [y]) \wedge \mathit{sorted}\ (y\ \#\ ys))\end{aligned} \tag{5.2}$$

$$\mathit{sorted}\ (x\ \#\ xs) \longrightarrow \mathit{sorted}\ xs \tag{5.3}$$

$$\mathit{sorted}\ (xs\ @\ [y]) \longrightarrow \mathit{sorted}\ xs \tag{5.4}$$

$$\begin{aligned}&\mathit{sorted}\ (xs\ @\ [a]) \Longrightarrow \mathit{ins\_list}\ x\ (xs\ @\ a\ \#\ ys) =\\ &\quad (\mathit{if}\ x < a\ \mathit{then}\ \mathit{ins\_list}\ x\ xs\ @\ a\ \#\ ys\ \mathit{else}\ xs\ @\ \mathit{ins\_list}\ x\ (a\ \#\ ys))\end{aligned} \tag{5.5}$$

$$\begin{aligned}&\mathit{sorted}\ (xs\ @\ a\ \#\ ys) \Longrightarrow \mathit{del\_list}\ x\ (xs\ @\ a\ \#\ ys) =\\ &\quad (\mathit{if}\ x < a\ \mathit{then}\ \mathit{del\_list}\ x\ xs\ @\ a\ \#\ ys\ \mathit{else}\ xs\ @\ \mathit{del\_list}\ x\ (a\ \#\ ys))\end{aligned} \tag{5.6}$$

$$\mathit{sorted}\ (x\ \#\ xs) = ((\forall y{\in}\mathit{set}\ xs.\ x < y) \wedge \mathit{sorted}\ xs) \tag{5.7}$$

$$\mathit{sorted}\ (xs\ @\ [x]) = (\mathit{sorted}\ xs \wedge (\forall y{\in}\mathit{set}\ xs.\ y < x)) \tag{5.8}$$

Fig. 5.2. Lemmas for *sorted*, *ins_list*, *del_list*

push *ins_list* inwards. Terms of the form $\mathit{del\_list}\ x\ (xs_1\ @\ a_1\ \#\ xs_2\ @\ a_2\ \#\ \ldots\ \#\ xs_n)$ are rewritten with equation (5.6) (and the defining equations for *del_list*) to push *del_list* inwards.

The *isin* property in Figure 5.1 can be proved with the help of (5.1), (5.7) and (5.8).

The lemmas in Figure 5.2 form the complete set of basic lemmas on which the automatic proofs of almost all search trees in the book rest; only splay trees need additional lemmas.

5.4.3 Sorted Lists Implement Sets

It remains to be shown that the list-based specification (Figure 5.1) implies the set-based correctness properties in Section 5.3. Remember that $\mathit{bst}\ t = \mathit{sorted}\ (\mathit{inorder}\ t)$. The correctness properties

$$\begin{aligned}&\mathit{set\_tree}\ \mathit{empty} = \{\}\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{set\_tree}\ (\mathit{insert}\ x\ t) = \mathit{set\_tree}\ t \cup \{x\}\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{set\_tree}\ (\mathit{delete}\ x\ t) = \mathit{set\_tree}\ t - \{x\}\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{isin}\ t\ x = (x \in \mathit{set\_tree}\ t)\end{aligned}$$

are a consequence of $\mathit{set\_tree}\ t = \mathit{set}\ (\mathit{inorder}\ t)$ (a basic tree lemma), the properties in Figure 5.1 and the inductive correctness properties

$$\begin{aligned}&\mathit{set}\ (\mathit{ins\_list}\ x\ xs) = \mathit{set}\ xs \cup \{x\}\\ &\mathit{sorted}\ xs \longrightarrow \mathit{set}\ (\mathit{del\_list}\ x\ xs) = \mathit{set}\ xs - \{x\}\end{aligned}$$

Preservation of the invariant

$$\begin{aligned}&\mathit{sorted}\ (\mathit{inorder}\ \mathit{empty})\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{sorted}\ (\mathit{inorder}\ (\mathit{insert}\ x\ t))\\ &\mathit{sorted}\ (\mathit{inorder}\ t) \longrightarrow \mathit{sorted}\ (\mathit{inorder}\ (\mathit{delete}\ x\ t))\end{aligned}$$

are trivial or consequences of the properties in Figure 5.1 and the preservation of *sorted* by *ins_list* and *del_list*:

$$\mathit{sorted}\ \mathit{xs} \longrightarrow \mathit{sorted}\ (\mathit{ins\_list}\ x\ \mathit{xs})$$
$$\mathit{sorted}\ \mathit{xs} \longrightarrow \mathit{sorted}\ (\mathit{del\_list}\ x\ \mathit{xs})$$

This means in particular that preservation of *sorted* ∘ *inorder* is guaranteed for any implementation of *empty*, *insert* and *delete* that satisfies the list-based specification in Figure 5.1.

5.5 Interval Trees [2]

In this section we study binary trees for representing a set of intervals, called interval trees. In addition to the usual insertion and deletion functions of standard BSTs, interval trees support a function for determining whether a given interval overlaps with some interval in the tree.

5.5.1 Augmented BSTs

The efficient implementation of the search for an overlapping interval relies on an additional piece of information in each node. Thus interval trees are another example of augmented trees as introduced in Section 4.4. We reuse the modified definitions of *set_tree* and *inorder* from that section. Moreover we use a slightly adjusted version of *isin*:

$\mathit{isin} :: ('a \times 'b)\ \mathit{tree} \Rightarrow 'a \Rightarrow \mathit{bool}$

$\mathit{isin}\ \langle\rangle\ \_ = \mathit{False}$
$\mathit{isin}\ \langle l,\ (a,\ \_),\ r\rangle\ x$
$= (\mathit{case}\ \mathit{cmp}\ x\ a\ \mathit{of}\ \mathit{LT} \Rightarrow \mathit{isin}\ l\ x \mid \mathit{EQ} \Rightarrow \mathit{True} \mid \mathit{GT} \Rightarrow \mathit{isin}\ r\ x)$

This works for any kind of augmented BST, not just interval trees.

5.5.2 Intervals

An interval *'a ivl* is simply a pair of lower and upper bound, accessed by functions *low* and *high*, respectively. Intuitively, an interval represents the closed set between *low* and *high*. The standard mathematical notation is $[l, h]$, the Isabelle notation is $\{l..h\}$. We restrict ourselves to non-empty intervals:

[2] isabelle/src/HOL/Data_Structures/Interval_Tree.thy

$low\ p \leqslant high\ p$

Type $'a$ can be any linearly ordered type with a minimum element $\bot$ (for example, the natural numbers or the real numbers extended with $-\infty$). Intervals can be linearly ordered by first comparing *low*, then comparing *high*. The definitions are as follows:

$(x < y) = (low\ x < low\ y \lor low\ x = low\ y \land high\ x < high\ y)$
$(x \leqslant y) = (low\ x < low\ y \lor low\ x = low\ y \land high\ x \leqslant high\ y)$

Two intervals overlap if they have at least one point in common:

$overlap\ x\ y = (low\ y \leqslant high\ x \land low\ x \leqslant high\ y)$

The reader should convince herself that *overlap* does what it is supposed to do: $overlap\ x\ y = (\{low\ x..high\ x\} \cap \{low\ y..high\ y\} \neq \{\})$

We also define the concept of an interval overlapping with some interval in a set:

$has\_overlap\ S\ y = (\exists\, x{\in}S.\ overlap\ x\ y)$

5.5.3 Interval Trees

An interval tree associates to each node a number *max_hi*, which records the maximum *high* value of all intervals in the subtree rooted at that node. This value is updated during insert and delete operations, and will be crucial for enabling efficient determination of overlap with some interval in the tree.

type_synonym $'a\ ivl\_tree = ('a\ ivl \times 'a)\ tree$

$max\_hi :: 'a\ ivl\_tree \Rightarrow 'a$
$max\_hi\ \langle\rangle = \bot$
$max\_hi\ \langle\_, (\_, m), \_\rangle = m$

If the *max_hi* value of every node in a tree agrees with *max*3

```
inv_max_hi :: 'a ivl_tree ⇒ bool
inv_max_hi ⟨⟩ = True
inv_max_hi ⟨l, (a, m), r⟩
= (inv_max_hi l ∧ inv_max_hi r ∧
   m = max3 a (max_hi l) (max_hi r))

max3 :: 'a ivl ⇒ 'a ⇒ 'a ⇒ 'a
max3 a m n = max (high a) (max m n)
```

it follows by induction that *max_hi* is the maximum value of *high* in the tree and comes from some node in the tree:

Lemma 5.1. $inv\_max\_hi\ t \wedge a \in set\_tree\ t \longrightarrow high\ a \leqslant max\_hi\ t$

Lemma 5.2.
$inv\_max\_hi\ t \wedge t \neq \langle\rangle \longrightarrow (\exists\, a{\in}set\_tree\ t.\ high\ a = max\_hi\ t)$

5.5.4 Implementing Sets of Intervals via Interval Trees

Interval trees can implement sets of intervals via unbalanced BSTs as in Section 5.2. Of course *empty* = ⟨⟩. Function *isin* was already defined in Section 5.5.1 Insertion and deletion are also very close to the versions in Section 5.2, but the value of *max_hi* must be computed (by *max3*) for each new node. We follow Section 4.4 and introduce a smart constructor *node* for that purpose and replace ⟨*l*, *a*, *r*⟩ by *node l a r* (on the right-hand side):

```
node :: 'a ivl_tree ⇒ 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
node l a r = ⟨l, (a, max3 a (max_hi l) (max_hi r)), r⟩

insert :: 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
insert x ⟨⟩ = ⟨⟨⟩, (x, high x), ⟨⟩⟩
insert x ⟨l, (a, m), r⟩ = (case cmp x a of
                            LT ⇒ node (insert x l) a r |
                            EQ ⇒ ⟨l, (a, m), r⟩ |
                            GT ⇒ node l a (insert x r))

split_min :: 'a ivl_tree ⇒ 'a ivl × 'a ivl_tree
split_min ⟨l, (a, _), r⟩
= (if l = ⟨⟩ then (a, r)
    else let (x, l') = split_min l in (x, node l' a r))

delete :: 'a ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ node (delete x l) a r |
    EQ ⇒ if r = ⟨⟩ then l else let (x, y) = split_min r in node l x y |
    GT ⇒ node l a (delete x r))
```

The correctness proofs for insertion and deletion cover two aspects. Functional correctness and preservation of the invariant $sorted \circ inorder$ (the BST property) are proved exactly as in Section 5.3 for ordinary BSTs. Preservation of the invariant inv_max_hi can be proved by a sequence of simple inductive properties. In the end the main correctness properties are

$$sorted\ (inorder\ t) \longrightarrow inorder\ (insert\ x\ t) = ins\_list\ x\ (inorder\ t)$$
$$sorted\ (inorder\ t) \longrightarrow inorder\ (delete\ x\ t) = del\_list\ x\ (inorder\ t)$$
$$inv\_max\_hi\ t \longrightarrow inv\_max\_hi\ (insert\ x\ t)$$
$$inv\_max\_hi\ t \longrightarrow inv\_max\_hi\ (delete\ x\ t)$$

Defining $invar\ t \equiv inv\_max\_hi\ t \wedge sorted\ (inorder\ t)$ we obtain the following top-level correctness corollaries:

$$invar\ s \longrightarrow set\_tree\ (insert\ x\ s) = set\_tree\ s \cup \{x\}$$
$$invar\ s \longrightarrow set\_tree\ (delete\ x\ s) = set\_tree\ s - \{x\}$$
$$invar\ s \longrightarrow invar\ (insert\ x\ s)$$
$$invar\ s \longrightarrow invar\ (delete\ x\ s)$$

5.5.5 Searching for an Overlapping Interval

The key added functionality of interval trees over ordinary BSTs is the search function. It searches for an overlapping rather than identical interval:

```
search :: 'a ivl_tree ⇒ 'a ivl ⇒ bool
search ⟨⟩ _ = False
search ⟨l, (a, _), r⟩ x
= (if overlap x a then True
    else if l ≠ ⟨⟩ ∧ low x ⩽ max_hi l then search l x else search r x)
```

The following theorem expresses the correctness of *search* assuming the same invariants as before; *bst t* would work just as well as *sorted (inorder t)*.

Theorem 5.3. $inv\_max\_hi\ t \wedge sorted\ (inorder\ t) \longrightarrow search\ t\ x = has\_overlap\ (set\_tree\ t)\ x$

Proof. The result is clear when t is $\langle\rangle$. Now suppose t is in the form $\langle l, (a, m), r\rangle$, where m is the value of max_hi at root. If a overlaps with x, search returns *True* as expected. Otherwise, there are two cases.

- If $l \neq \langle\rangle$ and $low\ x \leqslant max\_hi\ l$, the search goes to the left subtree. If there is an interval in the left subtree overlapping with x, then the search returns *True* as expected. Otherwise, we show there is also no interval in the right subtree overlapping with x. Since $l \neq \langle\rangle$, Lemma 5.2 yields a node p in the left subtree such that $high\ p = max\_hi\ l$. Since $low\ x \leqslant max\_hi\ l$, we have $low\ x \leqslant high\ p$. Since p does not overlap with x, we must have $high\ x < low\ p$. But then, for every interval rp in the right subtree, $low\ p \leqslant low\ rp$, so that $high\ x < low\ rp$, which implies that rp does not overlap with x.
- Now we consider the case where either $l = \langle\rangle$ or $max\_hi\ l < low\ x$. In this case, the search goes to the right. We show there is no interval in the left subtree that overlaps with x. This is clear if $l = \langle\rangle$. Otherwise, for each interval lp, we have $high\ lp \leqslant max\_hi\ l$ by Lemma 5.1, so that $high\ lp < low\ x$, which means lp does not overlap with x.

□

Exercise 5.3. In this exercise you need to define a function that determines if a given point is in some interval in a given interval tree. Starting with

$in\_ivl :: 'a \Rightarrow 'a\ ivl \Rightarrow bool$

$in\_ivl\ x\ iv = (low\ iv \leqslant x \wedge x \leqslant high\ iv)$

write a recursive function

$search1 :: 'a\ ivl\_tree \Rightarrow 'a \Rightarrow bool$

such that $search1\ x\ t$ is *True* iff there is some interval iv in t such that $in\_ivl\ x\ iv$. Prove

$inv\_max\_hi\ t \wedge bst\ t \longrightarrow search1\ t\ x = (\exists iv \in set\_tree\ t.\ in\_ivl\ x\ iv)$

5.5.6 Notes

While this section demonstrated how to augment an ordinary binary tree with intervals, any of the balanced binary trees (such as red-black tree) can be augmented in a similar manner. We leave this as exercises.

Interval trees have many applications in computational geometry. A basic example is as follows. Consider a set of rectangles whose sides are aligned to the x and y-axes. We wish to efficiently determine whether any pair of rectangles in the set intersect each other (i.e. sharing a point, including boundaries). This can be done using a "sweep line" algorithm as follows. For each rectangle $[x_l, x_h] \times [y_l, y_h]$, we create two events: insert interval $[x_l, x_h]$ at y-coordinate y_l and delete interval $[x_l, x_h]$ at y-coordinate y_h. Perform the events, starting from an empty interval tree, in order of y-coordinate, with insertion events performed before deletion events. At each insertion, check whether the interval to be inserted overlaps with any of the existing interval in the tree. If yes, we have found an intersection between two rectangles. If no overlap of intervals is detected throughout the process, then no pair of rectangles intersect. When using an interval tree based on a balanced binary tree, the time complexity of this procedure is $O(n \log n)$, where n is the number of rectangles.

We refer to [6, Section 14.3] for another exposition on interval trees and their applications. Interval trees, together with the application of finding rectangle intersection, have been formalized in [29].

6

Abstract Data Types

In the previous chapter we looked at a very specific example of an abstract data type, namely sets. In this chapter we consider abstract data types in general and in particular the model-oriented approach to the specification of abstract data types. This will lead to a generic format for such specifications.

6.1 Abstract Data Types

Abstract data types (ADTs) can be summarized by the following slogan:

$$\text{ADT} = \textit{interface} + \textit{specification}$$

where the interface lists the operations supported by the ADT and the specification describes the behaviour of these operations. For example, our set ADT has the following interface:

$\textit{empty} :: {}'s$
$\textit{insert} :: {}'a \Rightarrow {}'s \Rightarrow {}'s$
$\textit{delete} :: {}'a \Rightarrow {}'s \Rightarrow {}'s$
$\textit{isin} :: {}'s \Rightarrow {}'a \Rightarrow \textit{bool}$

The purpose of an ADT is to be able to write applications based on that ADT that will work with any implementation of the ADT. To that end one can prove properties of the application that are solely based on the specification of the ADT. That is, one can write generic algorithms and prove generic correctness theorems about them in the context of the ADT specification.

6.2 Model-Oriented Specifications

We follow the model-oriented style of specification [17]. In that style, an abstract type is specified by giving an abstract model for it. For simplicity we assume that each ADT describes one **type of interest** T. In the set interface T is $'s$. This type T must be specified by some existing HOL type A, the abstract model. In the case of sets this is straightforward: the model for sets is simply the HOL type $'a$ *set*. The motto is that T should behave like A. In order to bridge the gap between the two types, the specification needs an

- **abstraction function** $\alpha :: T \Rightarrow A$

that maps concrete values to their abstract counterparts. Moreover, in general only some elements of T represent elements of A. For example, in the set implementation in the previous chapter not all trees but only BSTs represent sets. Thus the specification should also take into account an

- **invariant** $invar :: T \Rightarrow bool$

Note that the abstraction function and the invariant are not part of the interface, but they are essential for specification and verification purposes.

As an example, the ADT of sets is shown in Figure 11.1 with suggestive keywords and a fixed mnemonic naming schema for the labels in the specification. This is the template for ADTs that we follow throughout the book.

ADT *Set* =

interface
$empty :: 's$
$insert :: 'a \Rightarrow 's \Rightarrow 's$
$delete :: 'a \Rightarrow 's \Rightarrow 's$
$isin :: 's \Rightarrow 'a \Rightarrow bool$

abstraction $set :: 's \Rightarrow 'a\ set$
invariant $invar :: 's \Rightarrow bool$

specification
$invar\ empty$ (*empty-inv*)
$set\ empty = \{\}$ (*empty*)
$invar\ s \longrightarrow invar\ (insert\ x\ s)$ (*insert-inv*)
$invar\ s \longrightarrow set(insert\ x\ s) = set\ s \cup \{x\}$ (*insert*)
$invar\ s \longrightarrow invar\ (delete\ x\ s)$ (*delete-inv*)
$invar\ s \longrightarrow set\ (delete\ x\ s) = set\ s - \{x\}$ (*delete*)
$invar\ s \longrightarrow isin\ s\ x = (x \in set\ s)$ (*isin*)

Fig. 6.1. ADT *Set*

We have intentionally refrained from showing the Isabelle formalization using so-called **locales** and have opted for a more intuitive textual format that is not Isabelle-specific, in accordance with the general philosophy of this book.

The actual Isabelle text can of course be found in the source files, and locales are explained in a dedicated manual [3].

We conclude this section by explaining what the specification of an arbitrary ADT looks like. We assume that for each function f of the interface there is a corresponding function f_A in the abstract model, i.e. defined on A. For a uniform treatment we extend α and *invar* to arbitrary types by setting $\alpha\ x = x$ and $invar\ x = True$ for all types other than T. Each function f of the interface gives rise to two properties in the specification: preservation of the invariant and simulation of f_A. The precondition is shared:

$$\begin{aligned}
&invar\ x_1 \wedge \ldots \wedge invar\ x_n \longrightarrow\\
&\quad invar(f\ x_1\ \ldots\ x_n) &&(f\text{-}inv)\\
&\quad \alpha(f\ x_1\ \ldots\ x_n) = f_A\ (\alpha\ x_1)\ \ldots\ (\alpha\ x_n) &&(f)
\end{aligned}$$

To understand how the specification of ADT *Set* is the result of this uniform schema one has to take two things into account:

- Precisely which abstract operations on type $'a\ set$ model the functions in the interface of the ADT *Set*? This correspondence is implicit in the specification: *empty* is modeled by $\{\}$, *insert* is modeled by $\lambda x\ s.\ s \cup \{x\}$, *delete* is modeled by $\lambda x\ s.\ s - \{x\}$ and *isin* is modeled by $\lambda s\ x.\ x \in s$.
- Because of the artificial extension of α and *invar* the above uniform format often collapses to something simpler where some α's and *invar*'s disappear.

TODO motivate by example why this spec helps to reason about clients?

6.3 Maps

An even more versatile data structure than sets are (efficient) maps from $'a$ to $'b$. In fact, sets can be viewed as maps from $'a$ to *bool*. Conversely, many data structures for sets also support maps, e.g. BSTs. Although, for simplicity, we focus on sets in this book, the ADT of maps should at least be introduced. It is shown in Figure 6.2. Type $'m$ is the type of maps from $'a$ to $'b$. The ADT *Map* is very similar to the ADT *Set* except that the abstraction function *lookup* is also part of the interface: it abstracts a map to a function of type $'a \Rightarrow\ 'b\ option$. This implies that the equations are between functions of that type. We use the function update notation (see Section 1.1.1) to explain *update* and *delete*: *update* is modeled by $\lambda m\ a\ b.\ m(a := b)$ and *delete* by $\lambda m\ a.\ m(a := None)$.

ADT *Map* =

interface
empty :: *'m*
update :: *'a* ⇒ *'b* ⇒ *'m* ⇒ *'m*
delete :: *'a* ⇒ *'m* ⇒ *'m*
lookup :: *'m* ⇒ *'a* ⇒ *'b option*

abstraction *lookup*
invariant *invar* :: *'m* ⇒ *bool*

specification

invar empty	(*empty-inv*)
lookup empty = (λ*a*. *None*)	(*empty*)
invar m ⟶ *invar* (*update a b m*)	(*update-inv*)
invar m ⟶ *lookup* (*update a b m*) = (*lookup m*)(*a* := *Some b*)	(*update*)
invar m ⟶ *invar* (*delete a m*)	(*delete-inv*)
invar m ⟶ *lookup* (*delete a m*) = (*lookup m*)(*a* := *None*)	(*delete*)

Fig. 6.2. ADT *Map*

6.4 Implementing ADTs

An implementation of an ADT consists of definitions for all the functions in the interface. If you want to verify the correctness of the implementation, you also need to provide definitions for the abstraction function and the invariant. The latter need not be executable unless they also occur in the interface and the implementation is meant to be executable. The abstraction function need not be surjective. For example, implementations of the ADT *Set* will normally only represent finite sets, e.g. by BSTs.

For Isabelle users: because ADTs are formalized as locales, an implementation of an ADT is an interpretation of the corresponding locale.

6.5 Exercises

Exercise 6.1. Modify the ADT *Set* specification by making *isin* the abstraction function (from *'s* to *'a* ⇒ *bool*). Follow the example of the ADT *Map* specification.

Exercise 6.2. In the ADT *Map* specification, following the general schema, there should be a property labeled (*lookup*), but it is missing. The reason is that given the correct abstract model of *lookup*, the equation becomes trivial: *invar m* ⟶ *lookup m a* = *lookup m a*. Why is that, which function models *lookup*?

Exercise 6.3. Modify the ADT *Map* as follows. Replace *update* and *delete* by a single function *modify* $:: 'a \Rightarrow \tau \Rightarrow 'm \Rightarrow 'm$ where τ is some suitable type that you need to figure out. Evaluating *modify a X m* should updated or delete the entry associated with a, depending on X and taking the old value associated with a into account. For example, if m maps $'a$ to *nat*, it should be possible to increment an association $a \mapsto n$ in m to $a \mapsto n{+}1$ by a single call *modify a inc m* for a suitable *inc*. What is more, if *lookup m a* $=$ *None* then *modify a inc m* should map a to 0. And if *lookup m a* $=$ *Some n* where $n \geqslant 99$ then *modify a inc m* should delete a from m. All three behaviours should be realized by the same *inc*.

Specify *modify* in the standard manner. In this context, define two functions *update* and *delete* with the help of *modify*. These definitions should be one-liners. Prove the *update* and *delete* properties in the original ADT *Map* as lemmas, using the definitions of *update* and *delete* and the specification of *modify*.

TODO (too hard?):

Exercise 6.4. Implement ADT *Map* via unbalanced BSTs (like *Set* in Chapter 5) using augmented trees. Verify the implementation by proving all the correctness properties in the specification of ADT *Map* directly, without any detour via sorted lists as in Section 5.4.

7

2-3 Trees [1]

This is the first in a series of chapters examining “balanced” search trees (as they are often called in the literature) where the height of the tree is logarithmic in its size and which can therefore be searched in logarithmic time. However, in most cases these trees are not *balanced* in our technical sense but obey some weaker invariant. TODO We will call them **logarithmic** rather than balanced to avoid confusion.

The most popular first example of logarithmic search trees are red-black trees. We start with 2-3 trees, where nodes can have 2 or 3 children, because red-black trees simulate 2-3 trees. We introduce red-black trees in the next chapter. The type of 2-3 trees is similar to the binary trees but with an additional constructor *Node3*:

```
datatype 'a tree23 =
  Leaf |
  Node2 ('a tree23) 'a ('a tree23) |
  Node3 ('a tree23) 'a ('a tree23) 'a ('a tree23)
```

The familiar syntactic sugar is sprinkled on top:

$$\begin{aligned} \langle\rangle &\equiv \mathit{Leaf} \\ \langle l,\ x,\ r\rangle &\equiv \mathit{Node2}\ l\ x\ r \\ \langle l,\ x,\ m,\ y,\ r\rangle &\equiv \mathit{Node3}\ l\ x\ m\ y\ r \end{aligned}$$

The size, height and the completeness of a 2-3 tree are defined by adding an equation for *Node3* to the corresponding definitions on binary trees:

[1] isabelle/src/HOL/Data_Structures/Tree23_Set.thy

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

$h\ \langle l, \_, m, \_, r\rangle = max\ (h\ l)\ (max\ (h\ m)\ (h\ r)) + 1$

$complete\ \langle l, \_, m, \_, r\rangle$
$= (complete\ l \wedge complete\ m \wedge complete\ r \wedge h\ l = h\ m \wedge$
$\quad h\ m = h\ r)$

A trivial induction yields $complete\ t \longrightarrow 2^{h\ t} \leqslant |t| + 1$: thus all operations on complete 2-3 trees have logarithmic complexity if they descend along a single branch and take constant time per node. This is the case and we will not discuss complexity in any more detail.

A nice property of 2-3 trees is that for every n there is a complete 2-3 tree of size n. As we will see below, completeness can be maintained under insertion and deletion in logarithmic time.

7.1 Implementation of ADT *Set*

The implementation will maintain the usual ordering invariant and completeness. When we speak of a 2-3 tree we will implicitly assume these two invariants now.

Searching a 2-3 tree is like searching a binary tree (see Section 5.2) but with one more defining equation:

```
isin ⟨l, a, m, b, r⟩ x
= (case cmp x a of LT ⇒ isin l x | EQ ⇒ True
   | GT ⇒ case cmp x b of LT ⇒ isin m x | EQ ⇒ True
               | GT ⇒ isin r x)
```

Insertion into a 2-3 tree must preserve the completeness invariant. Thus recursive calls must report back to the caller if the subtree has “overflown”, i.e. increased in height. Therefore insertion returns a result of type *'a upI*:

```
datatype 'a upI = TI ('a tree23) | OF ('a tree23) 'a ('a tree23)
```

This is the idea: If insertion into t returns

$TI\ t'$ then t and t' should have the same height,
$OF\ l\ x\ r$ then t and l and r should have the same height.

The insertion functions are shown in Figure 7.1. The actual work is performed by the recursive function *ins*. The element to be inserted is propagated down to a leaf, which causes an overflow of the leaf. If an overflow is returned from a recursive call it can be absorbed into a *Node2* but in a *Node3* it causes another overflow. At the root of the tree, function *treeI* converts values of type *'a upI* back into trees:

```
treeI :: 'a upI ⇒ 'a tree23
treeI (TI t) = t
treeI (OF l a r) = ⟨l, a, r⟩
```

```
insert x t = treeI (ins x t)

ins :: 'a ⇒ 'a tree23 ⇒ 'a upI
ins x ⟨⟩ = OF ⟨⟩ x ⟨⟩
ins x ⟨l, a, r⟩
= (case cmp x a of
    LT ⇒ case ins x l of TI l' ⇒ TI ⟨l', a, r⟩
              | OF l₁ b l₂ ⇒ TI ⟨l₁, b, l₂, a, r⟩
    | EQ ⇒ TI ⟨l, x, r⟩
    | GT ⇒ case ins x r of TI r' ⇒ TI ⟨l, a, r'⟩
               | OF r₁ b r₂ ⇒ TI ⟨l, a, r₁, b, r₂⟩)
ins x ⟨l, a, m, b, r⟩
= (case cmp x a of
    LT ⇒ case ins x l of TI l' ⇒ TI ⟨l', a, m, b, r⟩
              | OF l₁ c l₂ ⇒ OF ⟨l₁, c, l₂⟩ a ⟨m, b, r⟩
    | EQ ⇒ TI ⟨l, a, m, b, r⟩
    | GT ⇒ case cmp x b of
               LT ⇒ case ins x m of TI m' ⇒ TI ⟨l, a, m', b, r⟩
                         | OF m₁ c m₂ ⇒ OF ⟨l, a, m₁⟩ c ⟨m₂, b, r⟩
               | EQ ⇒ TI ⟨l, a, m, b, r⟩
               | GT ⇒ case ins x r of TI r' ⇒ TI ⟨l, a, m, b, r'⟩
                          | OF r₁ c r₂ ⇒ OF ⟨l, a, m⟩ b ⟨r₁, c, r₂⟩)
```

Fig. 7.1. Insertion into 2-3 tree

Deletion is dual. Recursive calls must report back to the caller if the subtree has "underflown", i.e. decreased in height. Therefore deletion returns a result of type *'a upD*:

```
datatype 'a upD = TD ('a tree23) | UF ('a tree23)
```

This is the idea: If deletion from t returns

$TD\ t'$ then t and t' should have the same height,
$UF\ t'$ then t should be one level higher than t'.

```
delete x t = treeD (del x t)

del :: 'a ⇒ 'a tree23 ⇒ 'a upD
del x ⟨⟩ = TD ⟨⟩
del x ⟨⟨⟩, a, ⟨⟩⟩ = (if x = a then UF ⟨⟩ else TD ⟨⟨⟩, a, ⟨⟩⟩)
del x ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩
= TD (if x = a then ⟨⟨⟩, b, ⟨⟩⟩
         else if x = b then ⟨⟨⟩, a, ⟨⟩⟩ else ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩)
del x ⟨l, a, r⟩
= (case cmp x a of LT ⇒ node21 (del x l) a r
    | EQ ⇒ let (a', r') = split_min r in node22 l a' r'
    | GT ⇒ node22 l a (del x r))
del x ⟨l, a, m, b, r⟩
= (case cmp x a of LT ⇒ node31 (del x l) a m b r
    | EQ ⇒ let (a', m') = split_min m in node32 l a' m' b r
    | GT ⇒ case cmp x b of LT ⇒ node32 l a (del x m) b r
                | EQ ⇒ let (b', r') = split_min r in node33 l a m b' r'
                | GT ⇒ node33 l a m b (del x r))

split_min :: 'a tree23 ⇒ 'a × 'a upD
split_min ⟨⟨⟩, a, ⟨⟩⟩ = (a, UF ⟨⟩)
split_min ⟨⟨⟩, a, ⟨⟩, b, ⟨⟩⟩ = (a, TD ⟨⟨⟩, b, ⟨⟩⟩)
split_min ⟨l, a, r⟩ = (let (x, l') = split_min l in (x, node21 l' a r))
split_min ⟨l, a, m, b, r⟩
= (let (x, l') = split_min l in (x, node31 l' a m b r))
```

Fig. 7.2. Deletion from 2-3 tree: main functions

The main deletion functions are shown in Figure 7.2. The actual work is performed by the recursive function *del*. If the element to be deleted is in a subtree, the result of a recursive call is reintegrated into the node via the auxiliary functions *nodeij* from Figure 7.3 that create a node with i subtrees where subtree j is given as an $'a\ upD$ value. If the element to be deleted is in the node itself, a replacement is obtained and deleted from a subtree via *split_min*. At the root of the tree, values of type *'a upD* are converted back into trees:

```
treeD :: 'a upD ⇒ 'a tree23
treeD (TD t) = t
treeD (UF t) = t
```

```
node21 :: 'a upD ⇒ 'a ⇒ 'a tree23 ⇒ 'a upD
node21 (TD t1) a t2 = TD ⟨t1, a, t2⟩
node21 (UF t1) a ⟨t2, b, t3⟩ = UF ⟨t1, a, t2, b, t3⟩
node21 (UF t1) a ⟨t2, b, t3, c, t4⟩ = TD ⟨⟨t1, a, t2⟩, b, ⟨t3, c, t4⟩⟩

node22 :: 'a tree23 ⇒ 'a ⇒ 'a upD ⇒ 'a upD
node22 t1 a (TD t2) = TD ⟨t1, a, t2⟩
node22 ⟨t1, b, t2⟩ a (UF t3) = UF ⟨t1, b, t2, a, t3⟩
node22 ⟨t1, b, t2, c, t3⟩ a (UF t4) = TD ⟨⟨t1, b, t2⟩, c, ⟨t3, a, t4⟩⟩

node31 :: 'a upD ⇒ 'a ⇒ 'a tree23 ⇒ 'a ⇒ 'a tree23 ⇒ 'a upD
node31 (TD t1) a t2 b t3 = TD ⟨t1, a, t2, b, t3⟩
node31 (UF t1) a ⟨t2, b, t3⟩ c t4 = TD ⟨⟨t1, a, t2, b, t3⟩, c, t4⟩
node31 (UF t1) a ⟨t2, b, t3, c, t4⟩ d t5
= TD ⟨⟨t1, a, t2⟩, b, ⟨t3, c, t4⟩, d, t5⟩

node32 :: 'a tree23 ⇒ 'a ⇒ 'a upD ⇒ 'a ⇒ 'a tree23 ⇒ 'a upD
node32 t1 a (TD t2) b t3 = TD ⟨t1, a, t2, b, t3⟩
node32 t1 a (UF t2) b ⟨t3, c, t4⟩ = TD ⟨t1, a, ⟨t2, b, t3, c, t4⟩⟩
node32 t1 a (UF t2) b ⟨t3, c, t4, d, t5⟩
= TD ⟨t1, a, ⟨t2, b, t3⟩, c, ⟨t4, d, t5⟩⟩

node33 :: 'a tree23 ⇒ 'a ⇒ 'a tree23 ⇒ 'a ⇒ 'a upD ⇒ 'a upD
node33 l a m b (TD r) = TD ⟨l, a, m, b, r⟩
node33 t1 a ⟨t2, b, t3⟩ c (UF t4) = TD ⟨t1, a, ⟨t2, b, t3, c, t4⟩⟩
node33 t1 a ⟨t2, b, t3, c, t4⟩ d (UF t5)
= TD ⟨t1, a, ⟨t2, b, t3⟩, c, ⟨t4, d, t5⟩⟩
```

Fig. 7.3. Deletion from 2-3 tree: auxiliary functions

7.2 Preservation of Completeness

As explained in Section 5.4 we do not go into the automatic functional correctness proofs but concentrate on the invariant preservation. To express the relationship between the height of a subtree before and after insertion we define the height function h on $'a\ upI$ values as follows:

$$h\ (TI\ t) = h\ t$$
$$h\ (OF\ l\ a\ r) = h\ l$$

This definition permits us to express the property $h\ (ins\ a\ t) = h\ t$ and prove by induction that

$$complete\ t \longrightarrow complete\ (treeI\ (ins\ a\ t)) \land h\ (ins\ a\ t) = h\ t$$

which implies by definition that

$$complete\ t \longrightarrow complete\ (insert\ a\ t)$$

The fact that deletion preserves completeness can be proved by a sequence of small lemmas. The appropriate definition of height on $'a\ upD$ values is the following one:

$$h\ (TD\ t) = h\ t$$
$$h\ (UF\ t) = h\ t + 1$$

This permits us to express $h\ (del\ x\ t) = h\ t$. We now list a sequence of properties that build on each other and culminate in completeness preservation of *delete*:

$$complete\ r \land complete\ (treeD\ l') \land h\ r = h\ l' \longrightarrow$$
$$complete\ (treeD\ (node21\ l'\ a\ r))$$
$$0 < h\ r \longrightarrow h\ (node21\ l'\ a\ r) = max\ (h\ l')\ (h\ r) + 1$$
$$split\_min\ t = (x,\ t') \land 0 < h\ t \land complete\ t \longrightarrow h\ t' = h\ t$$
$$split\_min\ t = (x,\ t') \land complete\ t \land 0 < h\ t \longrightarrow complete\ (treeD\ t')$$
$$complete\ t \longrightarrow h\ (del\ x\ t) = h\ t$$
$$complete\ t \longrightarrow complete\ (treeD\ (del\ x\ t))$$
$$complete\ t \longrightarrow complete\ (delete\ x\ t)$$

For each property of *node21* there are analogues properties for the other *nodeij* functions which we omit.

8 Red-Black Trees [1]

Red-black trees are a popular implementation technique for BSTs: the code is simpler than for 2-3 trees but guarantees logarithmic height too. The nodes are colored either red or black. Abstractly, red-black trees encode 2-3-4 trees where nodes have between 2 and 4 children. Each 2-3-4 node is encoded by a group of between 2 and 4 colored binary nodes as follows:

$$\begin{aligned}
\langle\rangle &\approx \langle\rangle \\
\langle A,a,B\rangle &\approx \langle A,a,B\rangle \\
\langle A,a,B,b,C\rangle &\approx \langle\langle A,a,B\rangle,b,C\rangle \text{ or } \langle A,a,\langle B,b,C\rangle\rangle \\
\langle A,a,B,b,C,c,D\rangle &\approx \langle\langle A,a,B\rangle,b,\langle C,c,D\rangle\rangle
\end{aligned}$$

Color expresses grouping: a black node is the root of 2-3-4 node, a red node is part of a bigger 2-3-4 node. Thus a red-black tree needs to satisfy the following properties or invariants:

1. The root is black.
2. Every $\langle\rangle$ is considered black.
3. If a node is red, its children are black.
4. All paths from a node to a leaf have the same number of black nodes.

The final property expresses that the corresponding 2-3-4 tree is complete. The last two properties imply that the tree has logarithmic height (see below).

We implement red-black trees as binary trees augmented (see Section 4.4) with a colour tag:

```
datatype color = Red | Black

type_synonym 'a rbt = ('a × color) tree
```

[1] isabelle/src/HOL/Data_Structures/RBT_Set.thy

Some new syntactic sugar is sprinkled on top:

$$R\ l\ a\ r \ \equiv\ \langle l,\ (a,\ \mathit{Red}),\ r\rangle$$
$$B\ l\ a\ r \ \equiv\ \langle l,\ (a,\ \mathit{Black}),\ r\rangle$$

The following functions get and set the color of a node:

$\mathit{color} :: \ 'a\ \mathit{rbt} \Rightarrow \mathit{color}$

$\mathit{color}\ \langle\rangle = \mathit{Black}$
$\mathit{color}\ \langle\_,\ (\_,\ c),\ \_\rangle = c$

$\mathit{paint} :: \mathit{color} \Rightarrow\ 'a\ \mathit{rbt} \Rightarrow\ 'a\ \mathit{rbt}$

$\mathit{paint}\ \_\ \langle\rangle = \langle\rangle$
$\mathit{paint}\ c\ \langle l,\ (a,\ \_),\ r\rangle = \langle l,\ (a,\ c),\ r\rangle$

The definition of *color* builds in the property that all leaves are black.

8.1 Invariants

The above informal description of the red-black tree invariants is formalized as the predicate *rbt* which (for reasons of modularity) is split into a color and a height invariant *invc* and *invh*:

$\mathit{rbt} :: \ 'a\ \mathit{rbt} \Rightarrow \mathit{bool}$

$\mathit{rbt}\ t = (\mathit{invc}\ t \wedge \mathit{invh}\ t \wedge \mathit{color}\ t = \mathit{Black})$

The color invariant expresses that red nodes must have black children:

$\mathit{invc} :: \ 'a\ \mathit{rbt} \Rightarrow \mathit{bool}$

$\mathit{invc}\ \langle\rangle = \mathit{True}$
$\mathit{invc}\ \langle l,\ (\_,\ c),\ r\rangle$
$= (\mathit{invc}\ l \wedge \mathit{invc}\ r \wedge$
$\quad (c = \mathit{Red} \longrightarrow \mathit{color}\ l = \mathit{Black} \wedge \mathit{color}\ r = \mathit{Black}))$

The height invariant expresses (via the **black height** *bh*) that all paths from the root to a leaf have the same number of black nodes:

```
invh :: 'a rbt ⇒ bool
invh ⟨⟩ = True
invh ⟨l, (_, _), r⟩ = (invh l ∧ invh r ∧ bh l = bh r)

bh :: 'a rbt ⇒ nat
bh ⟨⟩ = 0
bh ⟨l, (_, c), _⟩ = (if c = Black then bh l + 1 else bh l)
```

Note that although *bh* traverses only the left spine of the tree, the fact that *invh* traverses the complete tree ensures that all paths from the root to a leaf are considered. (See Exercise 8.1)

The split of the invariant into *invc* and *invh* improves modularity: frequently one can prove preservation of *invc* and *invh* separately, which facilitates proof search. For compactness we will mostly present the combined invariance properties.

8.1.1 Logarithmic Height

In a red-black tree, i.e. *rbt t*, every path from the root to a leaf has the same number of black nodes, and no such path has two red nodes in a row. Thus each leaf is at most twice as deep as any other leaf, and therefore $h\ t \leqslant 2 \cdot \lg |t|_1$. The detailed proof starts with the key inductive relationship between height and black height

$$invc\ t \wedge invh\ t \longrightarrow h\ t \leqslant 2 \cdot bh\ t + (\mathit{if}\ color\ t = Black\ \mathit{then}\ 0\ \mathit{else}\ 1)$$

which has the easy corollary $rbt\ t \longrightarrow h\ t\ /\ 2 \leqslant bh\ t$. Together with the easy inductive property

$$invc\ t \wedge invh\ t \longrightarrow 2^{bh\ t} \leqslant |t|_1$$

this implies $2\ powr\ (h\ t\ /\ 2) \leqslant 2\ powr\ bh\ t \leqslant |t|_1$ and thus $h\ t \leqslant 2 \cdot \lg |t|_1$ if *rbt t*.

8.2 Implementation of ADT *Set*

We implement sets by red-black trees that are also BSTs. As usual, we only discuss the proofs of preservation of the *rbt* invariant.

Function *isin* is implemented as for all augmented BSTs (see Section 5.5.1).

```
insert x t = paint Black (ins x t)

ins :: 'a ⇒ 'a rbt ⇒ 'a rbt
ins x ⟨⟩ = R ⟨⟩ x ⟨⟩
ins x (B l a r) = (case cmp x a of
                    LT ⇒ baliL (ins x l) a r |
                    EQ ⇒ B l a r |
                    GT ⇒ baliR l a (ins x r))
ins x (R l a r) = (case cmp x a of
                    LT ⇒ R (ins x l) a r |
                    EQ ⇒ R l a r |
                    GT ⇒ R l a (ins x r))

baliL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baliL (R (R t₁ a t₂) b t₃) c t₄ = R (B t₁ a t₂) b (B t₃ c t₄)
baliL (R t₁ a (R t₂ b t₃)) c t₄ = R (B t₁ a t₂) b (B t₃ c t₄)
baliL t₁ a t₂ = B t₁ a t₂

baliR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baliR t₁ a (R t₂ b (R t₃ c t₄)) = R (B t₁ a t₂) b (B t₃ c t₄)
baliR t₁ a (R (R t₂ b t₃) c t₄) = R (B t₁ a t₂) b (B t₃ c t₄)
baliR t₁ a t₂ = B t₁ a t₂
```

Fig. 8.1. Insertion into red-black tree

8.2.1 Insertion

Insertion is shown in Figure 8.1. The workhorse is function *ins*. It descends to the leaf where the element is inserted and it adjusts the colors on the way back up. The adjustment is performed by *baliL*/*baliR*. They combine arguments $l\ a\ r$ into a tree. If there is a red-red conflict in l/r, they rebalance and replace it by red-black. Inserting into a red node needs no immediate balancing because that will happen at the black node above it.

$$\begin{aligned}
&\mathit{ins}\ 1\ (B\ (R\ \langle\rangle\ 0\ \langle\rangle)\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle))\\
&= \mathit{baliL}\ (\mathit{ins}\ 1\ (R\ \langle\rangle\ 0\ \langle\rangle))\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle)\\
&= \mathit{baliL}\ (R\ \langle\rangle\ 0\ (\mathit{ins}\ 1\ \langle\rangle))\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle)\\
&= \mathit{baliL}\ (R\ \langle\rangle\ 0\ (R\ \langle\rangle\ 1\ \langle\rangle))\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle)\\
&= R\ (B\ \langle\rangle\ 0\ \langle\rangle)\ 1\ (B\ \langle\rangle\ 2\ (R\ \langle\rangle\ 3\ \langle\rangle))
\end{aligned}$$

Passing a red node up means an overflow occurred (as in 2-3 trees) that needs to be dealt with further up. At the latest at the very top where *insert* turns red into black.

Function *ins* preserves *invh* but not *invc*: it may return a tree with a red-red conflict at the top that is only resolved by an enclosing *baliL* or *baliR* (see example above). However, once the root node is colored black, everything is fine again. Thus we introduce the weaker invariant *invc2* as an abbreviation:

$$invc2\ t \equiv invc\ (paint\ Black\ t)$$

It is easy to prove that *baliL* and *baliR* preserve *invh* and upgrade from *invc2* to *invc*:

$$invh\ l \wedge invh\ r \wedge invc2\ l \wedge invc\ r \wedge bh\ l = bh\ r \longrightarrow$$
$$invc\ (baliL\ l\ a\ r) \wedge invh\ (baliL\ l\ a\ r) \wedge bh\ (baliL\ l\ a\ r) = bh\ l + 1$$
$$invh\ l \wedge invh\ r \wedge invc\ l \wedge invc2\ r \wedge bh\ l = bh\ r \longrightarrow$$
$$invc\ (baliR\ l\ a\ r) \wedge invh\ (baliR\ l\ a\ r) \wedge bh\ (baliR\ l\ a\ r) = bh\ l + 1$$

Another easy induction yields

$$invc\ t \wedge invh\ t \longrightarrow$$
$$invc2\ (ins\ x\ t) \wedge (color\ t = Black \longrightarrow invc\ (ins\ x\ t)) \wedge$$
$$invh\ (ins\ x\ t) \wedge bh\ (ins\ x\ t) = bh\ t$$

The corollary $rbt\ t \longrightarrow rbt\ (insert\ x\ t)$ is immediate.

8.2.2 Deletion

Deletion from a red-black tree is shown in Figure 8.2. It follows the deletion-by-replacing approach (Section 5.2.1). The tricky bit is how to maintain the invariants. As before, intermediate trees may only satisfy the weaker invariant *invc2*. Functions *del* and *split_min* decrease the black height of a tree with a black root node and leave the black height unchanged otherwise. To see that this makes sense, consider deletion from a singleton black or red node. The case that the element to be removed is not in the black tree can be dealt with by coloring the root node red. These are the precise input/output relations:

Lemma 8.1. $split\_min\ t = (x,\ t') \wedge t \neq \langle\rangle \wedge invh\ t \wedge invc\ t \longrightarrow$
$invh\ t' \wedge (color\ t = Red \longrightarrow bh\ t' = bh\ t \wedge invc\ t') \wedge$
$(color\ t = Black \longrightarrow bh\ t' = bh\ t - 1 \wedge invc2\ t')$

Lemma 8.2. $invh\ t \wedge invc\ t \wedge t' = del\ x\ t \longrightarrow$
$invh\ t' \wedge (color\ t = Red \longrightarrow bh\ t' = bh\ t \wedge invc\ t') \wedge$
$(color\ t = Black \longrightarrow bh\ t' = bh\ t - 1 \wedge invc2\ t')$

It is easy to see that the *del*-Lemma implies correctness of *delete*:

Corollary 8.3. $rbt\ t \longrightarrow rbt\ (delete\ x\ t)$

```
delete x t = paint Black (del x t)

del :: 'a ⇒ 'a rbt ⇒ 'a rbt
del _ ⟨⟩ = ⟨⟩
del x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ let l' = del x l
            in if l ≠ ⟨⟩ ∧ color l = Black then baldL l' a r else R l' a r |
    EQ ⇒ if r = ⟨⟩ then l
            else let (a', r') = split_min r
                 in if color r = Black then baldR l a' r' else R l a' r' |
    GT ⇒ let r' = del x r
            in if r ≠ ⟨⟩ ∧ color r = Black then baldR l a r' else R l a r')

split_min :: 'a rbt ⇒ 'a × 'a rbt
split_min ⟨l, (a, _), r⟩
= (if l = ⟨⟩ then (a, r)
    else let (x, l') = split_min l
         in (x, if color l = Black then baldL l' a r else R l' a r))

baldL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baldL (R t₁ a t₂) b t₃ = R (B t₁ a t₂) b t₃
baldL t₁ a (B t₂ b t₃) = baliR t₁ a (R t₂ b t₃)
baldL t₁ a (R (B t₂ b t₃) c t₄) = R (B t₁ a t₂) b (baliR t₃ c (paint Red t₄))
baldL t₁ a t₂ = R t₁ a t₂

baldR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
baldR t₁ a (R t₂ b t₃) = R t₁ a (B t₂ b t₃)
baldR (B t₁ a t₂) b t₃ = baliL (R t₁ a t₂) b t₃
baldR (R t₁ a (B t₂ b t₃)) c t₄ = R (baliL (paint Red t₁) a t₂) b (B t₃ c t₄)
baldR t₁ a t₂ = R t₁ a t₂
```

Fig. 8.2. Deletion from red-black tree

The proofs of the two lemmas need the following precise characterizations of *baldL* and *baldR*, the counterparts of *baliL* and *baliR*:

Lemma 8.4. $invh\ l \wedge invh\ r \wedge bh\ l + 1 = bh\ r \wedge invc2\ l \wedge invc\ r \wedge t' = baldL\ l\ a\ r \longrightarrow invh\ t' \wedge bh\ t' = bh\ r \wedge invc2\ t' \wedge (color\ r = Black \longrightarrow invc\ t')$

Lemma 8.5. $invh\ l \wedge invh\ r \wedge bh\ l = bh\ r + 1 \wedge invc\ l \wedge invc2\ r \wedge t' = baldR\ l\ a\ r \longrightarrow invh\ t' \wedge bh\ t' = bh\ l \wedge invc2\ t' \wedge (color\ l = Black \longrightarrow invc\ t')$

The proofs of these lemmas are by case analyses over the defining equations using the characteristic properties of *baliL* and *baliR* given above.

Proof. Lemma 8.2 is proved by induction on the computation of $del\ x\ t$. We concentrate on the induction step where $t = \langle l,\ (a,\ c),\ r\rangle$, and thus $invh\ l$, $invh\ r$, $bh\ l = bh\ r$, $invc\ l$ and $invc\ r$ because $invh\ t$ and $invc\ t$. Let $t' = del\ x\ t$. We need to prove $invh\ t'$ and

(R) $c = Red \longrightarrow bh\ t' = bh\ t \wedge invc\ t'$
(B) $c = Black \longrightarrow bh\ t' = bh\ t - 1 \wedge invc2\ t'$.

Consider the case $x < a$. Then $t' = baldL\ l'\ a\ r$ where $l' = del\ x\ l$. By IH we have $invh\ l'$, $color\ l = Red \longrightarrow bh\ l' = bh\ l \wedge invc\ l'$ and $color\ l = Black \longrightarrow bh\ l' = bh\ l - 1 \wedge invc2\ l'$. First we assume $l \neq \langle\rangle \wedge color\ l = Black$. Therefore $0 < bh\ l$ and hence $bh\ l' + 1 = bh\ r$ follows from $bh\ l' = bh\ l - 1$ and $bh\ l = bh\ r$. Thus all premises of Lemma 8.4 (where $l = l'$) hold and we obtain its conclusion: $invh\ t' \wedge bh\ t' = bh\ r \wedge invc2\ t' \wedge (color\ r = Black \longrightarrow invc\ t')$. It remains to prove (R) and (B). If $c = Red$ then $bh\ t' = bh\ r = bh\ l = bh\ t$ and $invc\ t$ implies $color\ r = Black$ and thus $invc\ t'$. If $c = Black$ then $bh\ t' = bh\ r = bh\ l = bh\ t - 1$ and $invc2\ t'$ follows from Lemma 8.4. Now we consider the case $\neg\ (l \neq \langle\rangle \wedge color\ l = Black)$. First assume $l = \langle\rangle$. Hence $t = \langle\langle\rangle,\ (a,\ c),\ r\rangle$ and $t' = R\ \langle\rangle\ a\ r$. Thus $invh\ t'$ follows from $invh\ t$. It remains to prove (R) and (B). If $c = Red$ then $bh\ t' = 0 = bh\ t$ and $invc\ t$ implies $color\ r = Black$ and thus $invc\ t'$ because $invc\ r$. If $c = Black$ then $bh\ t' = 0 = 1 - 1 = bh\ t - 1$ and $invc2\ t'$ follows from $invc\ r$. Now assume $color\ l \neq Black$. This implies $c = Black$ because of $invc\ t$. Hence $t' = R\ l'\ a\ r$ and $bh\ l' = bh\ l \wedge invc\ l'$ (by IH). Therefore $invh\ t' = (invh\ l' \wedge invh\ r \wedge bh\ l' = bh\ r)$ — the first two propositions have been derived above and the last proposition is easy: $bh\ l' = bh\ l = bh\ r$. It remains to prove (B) because $c = Black$. Proposition (B) follows easily: $bh\ t' = bh\ l' = bh\ l = bh\ t - 1$ and $invc2\ t' = (invc\ l' \wedge invc\ r)$ where both conjuncts have been derived above.

The case $a < x$ is analogous and the case $x = a$ is similar but simpler. □

The proof of Lemma 8.1 is similar but simpler.

8.2.3 Deletion by Joining

The basic idea was exemplified in the context of ordinary BSTs in Section 5.2.1. The code for red-black trees is shown in Figure 8.3: compared to Figure 8.2, the *EQ* case of *del* has changed and *join* is new.

Invariant preservation is proved much like before except that instead of *split_min* we now have *join* to take care of. The characteristic lemma:

```
del :: 'a ⇒ 'a rbt ⇒ 'a rbt
del _ ⟨⟩ = ⟨⟩
del x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ if l ≠ ⟨⟩ ∧ color l = Black then baldL (del x l) a r
          else R (del x l) a r |
    EQ ⇒ join l r |
    GT ⇒ if r ≠ ⟨⟩ ∧ color r = Black then baldR l a (del x r)
          else R l a (del x r))

join :: 'a rbt ⇒ 'a rbt ⇒ 'a rbt
join ⟨⟩ t = t
join t ⟨⟩ = t
join (R t₁ a t₂) (R t₃ c t₄)
= (case join t₂ t₃ of
    R u₂ b u₃ ⇒ R (R t₁ a u₂) b (R u₃ c t₄) |
    t₂₃ ⇒ R t₁ a (R t₂₃ c t₄)
join (B t₁ a t₂) (B t₃ c t₄)
= (case join t₂ t₃ of
    R u₂ b u₃ ⇒ R (B t₁ a u₂) b (B u₃ c t₄) |
    t₂₃ ⇒ baldL t₁ a (R t₂₃ c t₄)
join t₁ (R t₂ a t₃) = R (join t₁ t₂) a t₃ |
join (R t₁ a t₂) t₃ = R t₁ a (join t₂ t₃)
```

Fig. 8.3. Deletion from red-black tree by combining children

Lemma 8.6. $invh\ l \wedge invh\ r \wedge bh\ l = bh\ r \wedge invc\ l \wedge invc\ r \wedge$
$t' = join\ l\ r \longrightarrow$
$invh\ t' \wedge bh\ t' = bh\ l \wedge invc2\ t' \wedge$
$(color\ l = Black \wedge color\ r = Black \longrightarrow invc\ t')$

8.3 Exercises

Exercise 8.1. We already discussed informally why the definition of *invh* captures "all paths from the root to a leaf have the same number of black nodes" although *bh* only traverses the left spine. This exercises formalizes that discussion. The following function computes the set of black heights (number of black nodes) of all paths:

```
bhs :: 'a rbt ⇒ nat set
bhs ⟨⟩ = {0}
bhs ⟨l, (_, c), r⟩
```

$= (\mathit{let}\ H = \mathit{bhs}\ l \cup \mathit{bhs}\ r\ \mathit{in}\ \mathit{if}\ c = \mathit{Black}\ \mathit{then}\ \mathit{Suc}\ `\ H\ \mathit{else}\ H)$

where the infix operator (`) is predefined as $f\ `\ A = \{y \mid \exists x{\in}A.\ y = f\ x\}$. Prove $\mathit{invh}\ t \longleftrightarrow \mathit{bhs}\ t = \{\mathit{bh}\ t\}$. The direction $\mathit{invh}\ t \longrightarrow \mathit{bhs}\ t = \{\mathit{bh}\ t\}$ should be easy, the other direction should need some lemmas.

Exercise 8.2. TODO Modify delete: check left AND richt for Leaf before calling delmin

Exercise 8.3. TODO Linear time construction of RBTs from list

Exercise 8.4. Give a detailed informal proof of Lemma 8.6 in the style of the proof of Lemma 8.2.

Exercise 8.5. TODO (Bohua?) Rebase the theory of interval trees on red-black trees.

8.4 Bibliographic Remarks

Red-Black trees where invented by Bayer [4] who called them "symmetric binary B-trees". The red-black color convention was introduced by Guibas and Sedgewick [13] who studied their properties in greater depth. The first functional version of red-black trees is due to Okasaki [26] and everybody follows his code. A functional version of deletion was first given by Kahrs [18][2] and Section 8.2.3 is based on it. Germane [12] presents a function for deletion by replacement that is quite different from the one in Section 8.2.2. Our starting point were Isabelle proofs by Reiter and Krauss (based on Kahrs). Other verifications of red-black trees are reported by Filliâtre and Letouzey [10] (using their own deletion function) and Appel [2] (based on Kahrs).

[2] The code for deletion is not in the article but can be retrieved from this URL: `http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html`

9

AVL Trees [1]

The AVL tree [1] (named after its inventors Adelson-Velsky and Landis) is the granddaddy of efficient binary search trees. Its logarithmic height is maintained by rotating subtrees based on their height. For efficiency reasons the height of each subtree is stored in its root node. That is, the underlying data structure is a height-augmented tree (see Section 4.4):

```
type_synonym 'a tree_ht = ('a × nat) tree
```

Function *ht* extracts the height field and *node* is a smart constructor that sets the height field:

```
ht :: 'a tree_ht ⇒ nat
ht ⟨⟩ = 0
ht ⟨_, (_, n), _⟩ = n

node :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
node l a r = ⟨l, (a, max (ht l) (ht r) + 1), r⟩
```

An **AVL tree** is a tree that satisfies the AVL invariant: the height of the left and right child of any node differ by at most 1 (and the height field contains the correct value):

```
avl :: 'a tree_ht ⇒ bool
avl ⟨⟩ = True
```

[1] isabelle/src/HOL/Data_Structures/AVL_Set.thy

$$\begin{aligned}&avl\ \langle l,\ (a,\ n),\ r\rangle\\ &= (|int\ (h\ l) - int\ (h\ r)| \leqslant 1 \wedge\\ &\quad n = max\ (h\ l)\ (h\ r) + 1 \wedge avl\ l \wedge avl\ r)\end{aligned}$$

The conversion function $int :: nat \Rightarrow int$ is required because on natural numbers $0 - n = 0$.

9.1 Logarithmic Height

AVL trees have logarithmic height. The key insight for the proof is that M n, the minimal number of leaves of an AVL tree of height n, satisfies the recurrence relation $M\ (n + 2) = M\ (n + 1) + M\ n$. Instead of formalizing this function M we prove directly that an AVL tree of height n has at least $fib\ (n{+}2)$ leaves where fib is the Fibonacci function:

$$\begin{aligned}&fib :: nat \Rightarrow nat\\ &fib\ 0 = 0\\ &fib\ 1 = 1\\ &fib\ (n + 2) = fib\ (n + 1) + fib\ n\end{aligned}$$

Lemma 9.1. $avl\ t \longrightarrow fib\ (h\ t + 2) \leqslant |t|_1$

Proof. The proof is by induction on t. We focus on the induction step $t = \langle l,$ $(a,\ n),\ r\rangle$. and assume $avl\ t$; thus the IHs reduce to $fib\ (h\ l + 2) \leqslant |l|_1$ and $fib\ (h\ r + 2) \leqslant |r|_1$. We prove $fib\ (max\ (h\ l)\ (h\ r) + 3) \leqslant |l|_1 + |r|_1$, from which $avl\ t \longrightarrow fib\ (h\ t + 2) \leqslant |t|_1$ follows directly. There are two cases. We focus on $h\ r \leqslant h\ l$, $h\ l < h\ r$ is dual.

$$\begin{aligned}&fib\ (max\ (h\ l)\ (h\ r) + 3) = fib\ (h\ l + 3)\\ &= fib\ (h\ l + 2) + fib\ (h\ l + 1)\\ &\leqslant |l|_1 + fib\ (h\ l + 1) && \text{by } fib\ (h\ l + 2) \leqslant |l|_1\\ &\leqslant |l|_1 + |r|_1 && \text{by } fib\ (h\ r + 2) \leqslant |r|_1\end{aligned}$$

The last step is justified because $h\ l + 1 \leqslant h\ r + 2$ (which follows from $avl\ t$) and fib is monotone. □

Now we prove a well-known exponential lower bound for fib where $\varphi \equiv$ $(1 + \sqrt{5})\ /\ 2$:

Lemma 9.2. $\varphi^n \leqslant fib\ (n + 2)$

Proof. The proof is by induction on n by *fib* computation induction. The case $n = 0$ is trivial and the case $n = 1$ is easy. Now consider the induction step:

$$\begin{aligned} &\mathit{fib}\ (n + 2 + 2) = \mathit{fib}\ (n + 2 + 1) + \mathit{fib}\ (n + 2) \\ &\geqslant \varphi^{n + 1} + \varphi^{n} && \text{by IHs} \\ &= (\varphi + 1) \cdot \varphi^{n} \\ &= \varphi^{n + 2} && \text{because } \varphi + 1 = \varphi^2 \quad \square \end{aligned}$$

Combining the two lemmas yields $\mathit{avl}\ t \longrightarrow \varphi^{h\ t} \leqslant |t|_1$ and thus

Corollary 9.3. $\mathit{avl}\ t \longrightarrow h\ t \leqslant 1\ /\ \lg\ \varphi \cdot \lg\ |t|_1$

That is, the height of an AVL tree is at most $1\ /\ \lg\ \varphi \approx 1.44$ times worse than the optimal $\lg\ |t|_1$.

9.2 Implementation of ADT *Set*

9.2.1 Insertion

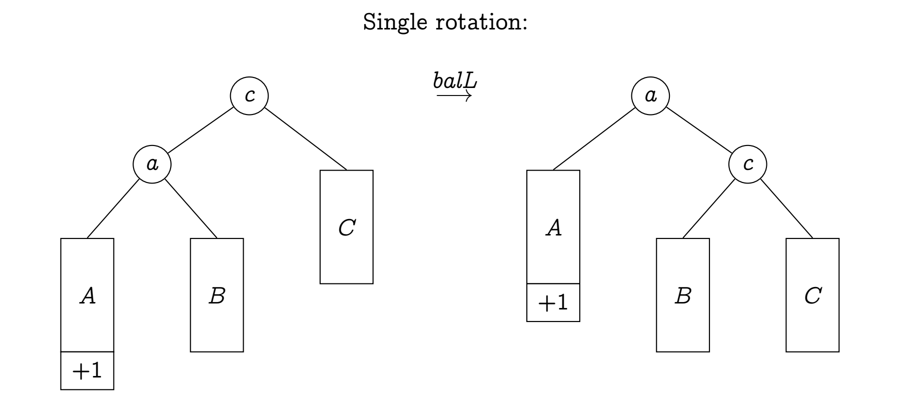
Insertion follows the standard approach: insert the element as usual and reestablish the AVL invariant on the way back up.

```
insert :: 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
insert x ⟨⟩ = ⟨⟨⟩, (x, 1), ⟨⟩⟩
insert x ⟨l, (a, n), r⟩ = (case cmp x a of
                              LT ⇒ balL (insert x l) a r |
                              EQ ⇒ ⟨l, (a, n), r⟩ |
                              GT ⇒ balR l a (insert x r))
```

Functions *balL*/*balR* readjust the tree after an insertion into the left/right child. The AVL invariant has been lost if the difference in height has become 2 — it cannot become more because the height can only increase by 1. We consider the definition of *balL* in Figure 9.1 (*balR* in Figure 9.2 is dual). If the AVL invariant has not been lost, i.e. if $ht\ AB \neq ht\ C + 2$ we can just return the AVL tree $node\ AB\ c\ C$. But if $ht\ AB = ht\ C + 2$, we need to “rotate” the subtrees suitably. Clearly AB must be of the form $\langle A, (ab, \_), B\rangle$. There are two cases, which are illustrated in Figure 9.1. Rectangles denote whole trees. Rectangles of the same height denote trees of the same height. Rectangles with a +1 denote the additional level due to insertion of the new element.

If $ht\ B \leqslant ht\ A$ then *balL* performs what is known as a single rotation.

```
balL :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
balL AB c C
= (if ht AB = ht C + 2
    then case AB of
           ⟨A, (a, x), B⟩ ⇒
             if ht B ⩽ ht A then node A a (node B c C)
             else case B of
                    ⟨B₁, (b, x), B₂⟩ ⇒ node (node A a B₁) b (node B₂ c C)
    else node AB c C)
```



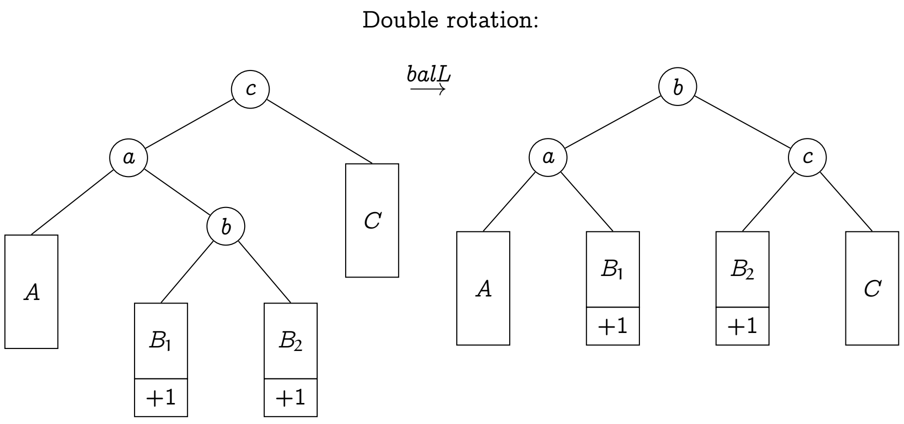


Fig. 9.1. Function *balL*

If $ht\ A < ht\ B$ then B must be of the form $\langle B_1, (b, \_), B_2\rangle$ (where either B_1 or B_2 has reached the same height as A) and *balL* performs what is known as a double rotation.

It is easy to check that in both cases the tree on the right satisfies the AVL invariant.

```
balR :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
balR A a BC
= (if ht BC = ht A + 2
    then case BC of
           ⟨B, (c, x), C⟩ ⇒
             if ht B ≤ ht C then node (node A a B) c C
             else case B of
                    ⟨B₁, (b, x), B₂⟩ ⇒ node (node A a B₁) b (node B₂ c C)
    else node A a BC)
```

Fig. 9.2. Function *balR*

Preservation of *avl* by *insert* cannot be proved in isolation but needs to be proved simultaneously with how *insert* changes the height (because *avl* depends on the height and *insert* requires *avl* for correct behaviour):

Theorem 9.4. $avl\ t \longrightarrow avl\ (insert\ x\ t) \wedge h\ (insert\ x\ t) \in \{h\ t,\ h\ t + 1\}$

The proof is by induction on t followed by a complete case analysis (which Isabelle automates).

9.2.2 Deletion

Figure 9.3 shows deletion by replacing (see 5.2.1). The recursive calls are dual to insertion: in terms of the difference in height, deletion of some element from one subtree is the same as insertion of some element into the other subtree. Thus functions *balR*/*balL* can again be employed to restore the invariant.

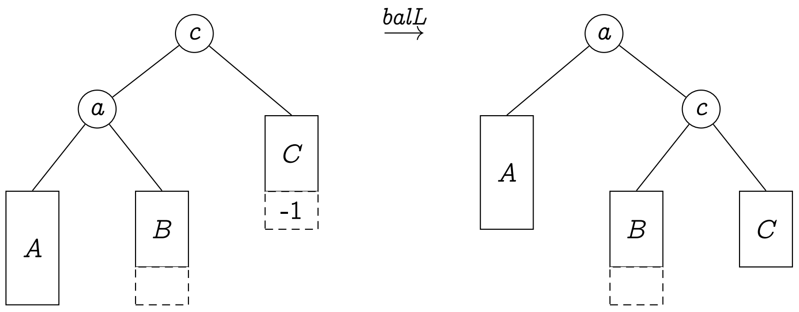
An element is deleted from a node by replacing it with the maximal element of the left subtree (the minimal element of the right subtree would work just as well). Function *split_max* performs that extraction and uses *balL* to restore the invariant after splitting an element off the right subtree.

The fact that *balR*/*balL* can be reused for deletion can be illustrated by drawing the corresponding rotation diagrams. We look at how the code for *balL* behaves when an element has been deleted from C. Dashed rectangles indicate a single additional level that may or may not be there. The label -1 indicates that the level has disappeared due to deletion.

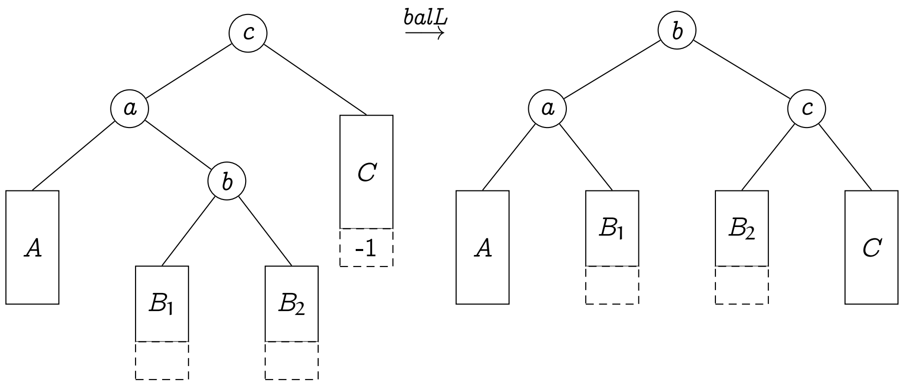
```
delete :: 'a ⇒ 'a tree_ht ⇒ 'a tree_ht
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, _), r⟩
= (case cmp x a of
    LT ⇒ balR (delete x l) a r |
    EQ ⇒ if l = ⟨⟩ then r else let (l', a') = split_max l in balR l' a' r |
    GT ⇒ balL l a (delete x r))

split_max :: 'a tree_ht ⇒ 'a tree_ht × 'a
split_max ⟨l, (a, _), r⟩
= (if r = ⟨⟩ then (l, a)
    else let (r', a') = split_max r in (balL l a r', a'))
```

Fig. 9.3. Deletion from AVL tree

Single rotation in *balL* after deletion in C:



Double rotation in *balL* after deletion in C:



At least one of B_1 and B_2 must have the same height as A.

Preservation of *avl* by *delete* can be proved in the same manner as for *insert* but we provide more of the details (partly because our Isabelle proof is less automatic). The following lemmas express *avl*-preservation by the auxiliary functions:

$$avl\ l \wedge avl\ r \wedge h\ r - 1 \leqslant h\ l \wedge h\ l \leqslant h\ r + 2 \longrightarrow \\ avl\ (balL\ l\ a\ r) \tag{9.1}$$

$$avl\ l \wedge avl\ r \wedge h\ l - 1 \leqslant h\ r \wedge h\ r \leqslant h\ l + 2 \longrightarrow \\ avl\ (balR\ l\ a\ r) \tag{9.2}$$

$$avl\ t \wedge t \neq \langle\rangle \longrightarrow \\ avl\ (fst\ (split\_max\ t)) \wedge \\ h\ t \in \{h\ (fst\ (split\_max\ t)),\ h\ (fst\ (split\_max\ t)) + 1\} \tag{9.3}$$

The first two are proved by the obvious cases analyses, the last one also requires induction.

As for *insert*, preservation of *avl* by *delete* needs to be proved simultaneously with how *delete* changes the height:

Theorem 9.5. $avl\ t \wedge t' = delete\ x\ t \longrightarrow avl\ t' \wedge h\ t \in \{h\ t',\ h\ t' + 1\}$

Proof. The proof is by induction on t followed by the case analyses dictated by the code for *delete*. We sketch the induction step. Let $t = \langle l, (a, n), r\rangle$ and $t' = delete\ x\ t$ and assume the IHs and $avl\ t$. The claim $avl\ t'$ follows using (9.1), (9.2) and (9.3). The claim $h\ t \in \{h\ t',\ h\ t' + 1\}$ can be proved directly from the definitions of *balL* and *balR*. □

9.3 Exercises

Exercise 9.1. The logarithmic height of AVL trees can be proved directly. Prove

$$avl\ t \wedge h\ t = n \longrightarrow 2^{n\ div\ 2} \leqslant |t|_1$$

by *fib* computation induction on n. This implies $avl\ t \longrightarrow h\ t \leqslant 2 \cdot \lg |t|_1$.

Exercise 9.2. Fibonacci trees are defined in analogy to Fibonacci numbers:

$$ft :: nat \Rightarrow unit\ tree$$

$$ft\ 0 = \langle\rangle \\ ft\ 1 = \langle\langle\rangle, (), \langle\rangle\rangle \\ ft\ (n + 2) = \langle ft\ (n + 1), (), ft\ n\rangle$$

We are only interested in the shape of these trees. Therefore the nodes just contain dummy unit values (). Hence we need to define the AVL invariant again for trees without annotations:

$avl0 :: 'a\ tree \Rightarrow bool$

$avl0\ \langle\rangle = True$
$avl0\ \langle l,\ \_,\ r\rangle = (|h\ l - h\ r| \leqslant 1 \wedge avl0\ l \wedge avl0\ r)$

Prove the following properties of Fibonacci trees:

$avl0\ (ft\ n) \qquad |ft\ n|_1 = fib\ (n + 2)$

Conclude that the Fibonacci trees are minimal (w.r.t. their size) among all AVL trees of a given height:

$avl\ t \longrightarrow |ft\ (h\ t)|_1 \leqslant |t|_1$

Exercise 9.3. Show that every balanced tree is an AVL tree:

$balanced\ t \longrightarrow avl0\ t$

As in the previous exercise we consider trees without height annotations.

Exercise 9.4. Generalize AVL trees so **height-balanced trees** where the condition

$|int\ (h\ l) - int\ (h\ r)| \leqslant 1$

in the invariant is replaced by

$|int\ (h\ l) - int\ (h\ r)| \leqslant m$

where $m \geqslant 1$ is some fixed integer. Modify the invariant and the insertion and deletion functions and prove that the latter fulfill the same correctness theorems as before. You do not need to prove the logarithmic height of height-balanced trees.

9.4 An Optimization [2]

Instead of recording the height of the tree in each node, it suffices to record the **balance factor**, i.e. the difference in height of its two children. Rather than the three integers $\{-1, 0, 1\}$ we utilize a new datatype:

datatype $bal = Lh \mid Bal \mid Rh$

type_synonym $'a\ tree\_bal = ('a \times bal)\ tree$

The names Lh and Rh stand for "left-heavy" and "right-heavy". The AVL invariant for these trees reflect these names:

[2] isabelle/src/HOL/Data_Structures/AVL_Bal_Set.thy

```
avl :: 'a tree_bal ⇒ bool
avl ⟨⟩ = True
avl ⟨l, (_, b), r⟩ = ((case b of
                         Lh ⇒ h l = h r + 1 |
                         Bal ⇒ h r = h l |
                         Rh ⇒ h r = h l + 1) ∧
                       avl l ∧ avl r)
```

The code for insertion (and deletion) is similar to the height-based version. The key difference is that the test if AVL invariant as been lost cannot be based on the height anymore. We need to detect if the tree has increased in height upon insertion based on the balance factors. The key insight is that a height increase is coupled with a change from *Bal* to *Lh* or *Rh*. Except when we transition from $\langle\rangle$ to $\langle\langle\rangle, (a, Bal), \langle\rangle\rangle$. This insight is encoded in the test *incr*:

```
is_bal :: 'a tree_bal ⇒ bool
is_bal ⟨_, (_, b), _⟩ = (b = Bal)

incr :: 'a tree_bal ⇒ 'b tree_bal ⇒ bool
incr t t' = (t = ⟨⟩ ∨ is_bal t ∧ ¬ is_bal t')
```

The test for a height increase compares the trees before and after insertion. Therefore it has been pulled out of the balance functions into insertion:

```
insert :: 'a ⇒ 'a tree_bal ⇒ 'a tree_bal
insert x ⟨⟩ = ⟨⟨⟩, (x, Bal), ⟨⟩⟩
insert x ⟨l, (a, b), r⟩
= (case cmp x a of
    LT ⇒ let l' = insert x l
          in if incr l l' then balL l' a b r else ⟨l', (a, b), r⟩ |
    EQ ⇒ ⟨l, (a, b), r⟩ |
    GT ⇒ let r' = insert x r
          in if incr r r' then balR l a b r' else ⟨l, (a, b), r'⟩)
```

The balance functions are shown in Figure 9.4. Function *rot2* implements double rotations.

Function *balL* is called if the left subtree *AB* has increased in height. If the tree was *Lh* then single or double rotations are necessary to restore balance.

```
balL :: 'a tree_bal ⇒ 'a ⇒ bal ⇒ 'a tree_bal ⇒ 'a tree_bal
balL AB c bc C
= (case bc of
    Lh ⇒ case AB of
           ⟨A, (a, Lh), B⟩ ⇒ ⟨A, (a, Bal), ⟨B, (c, Bal), C⟩⟩ |
           ⟨A, (a, Bal), B⟩ ⇒ ⟨A, (a, Rh), ⟨B, (c, Lh), C⟩⟩ |
           ⟨A, (a, Rh), B⟩ ⇒ rot2 A a B c C |
    Bal ⇒ ⟨AB, (c, Lh), C⟩ |
    Rh ⇒ ⟨AB, (c, Bal), C⟩)

balR :: 'a tree_bal ⇒ 'a ⇒ bal ⇒ 'a tree_bal ⇒ 'a tree_bal
balR A a ba BC
= (case ba of
    Lh ⇒ ⟨A, (a, Bal), BC⟩ |
    Bal ⇒ ⟨A, (a, Rh), BC⟩ |
    Rh ⇒ case BC of
           ⟨B, (c, Lh), C⟩ ⇒ rot2 A a B c C |
           ⟨B, (c, Bal), C⟩ ⇒ ⟨⟨A, (a, Rh), B⟩, (c, Lh), C⟩ |
           ⟨B, (c, Rh), C⟩ ⇒ ⟨⟨A, (a, Bal), B⟩, (c, Bal), C⟩)

rot2 :: 'a tree_bal ⇒ 'a ⇒ 'a tree_bal ⇒ 'a ⇒ 'a tree_bal ⇒ 'a tree_bal
rot2 A a B c C
= (case B of
    ⟨B₁, (b, bb), B₂⟩ ⇒
      let b₁ = if bb = Rh then Lh else Bal; b₂ = if bb = Lh then Rh else Bal
      in ⟨⟨A, (a, b₁), B₁⟩, (b, Bal), ⟨B₂, (c, b₂), C⟩⟩)
```

Fig. 9.4. Functions *balL* and *balR*

Otherwise we simply need to adjust the balance factors. Function *balR* is dual to *balL*.

For deletion we need to test if the height has decreased and *decr* implements this test:

```
decr :: 'a tree_bal ⇒ 'b tree_bal ⇒ bool
decr t t' = (t ≠ ⟨⟩ ∧ (t' = ⟨⟩ ∨ ¬ is_bal t ∧ is_bal t'))
```

The functions *incr* and *decr* are almost dual except that *incr* implicitly assumes $t' \neq \langle\rangle$ because insertion is guaranteed to return a *Node*. Thus we could use *decr* instead of *incr* but not the other way around.

Deletion and *split_max* change in the same manner as insertion:

```
delete :: 'a ⇒ 'a tree_bal ⇒ 'a tree_bal
delete _ ⟨⟩ = ⟨⟩
delete x ⟨l, (a, ba), r⟩
= (case cmp x a of
    LT ⇒ let l' = delete x l
            in if decr l l' then balR l' a ba r else ⟨l', (a, ba), r⟩
  | EQ ⇒ if l = ⟨⟩ then r
             else let (l', a') = split_max l
                    in if decr l l' then balR l' a' ba r else ⟨l', (a', ba), r⟩
  | GT ⇒ let r' = delete x r
             in if decr r r' then balL l a ba r' else ⟨l, (a, ba), r'⟩)

split_max :: 'a tree_bal ⇒ 'a tree_bal × 'a
split_max ⟨l, (a, ba), r⟩
= (if r = ⟨⟩ then (l, a)
    else let (r', a') = split_max r;
            t' = if decr r r' then balL l a ba r' else ⟨l, (a, ba), r'⟩
         in (t', a'))
```

In the end we have the following correctness theorems:

Theorem 9.6. $\mathit{avl}\ t \wedge t' = \mathit{insert}\ x\ t \longrightarrow$
$\mathit{avl}\ t' \wedge h\ t' = h\ t + (\mathit{if}\ \mathit{incr}\ t\ t'\ \mathit{then}\ 1\ \mathit{else}\ 0)$

This theorem tells us not only that *avl* is preserved but also that *incr* indicates correctly if the height has increased or not.

Similarly for deletion and *decr*:

Theorem 9.7. $\mathit{avl}\ t \wedge t' = \mathit{delete}\ x\ t \longrightarrow$
$\mathit{avl}\ t' \wedge h\ t = h\ t' + (\mathit{if}\ \mathit{decr}\ t\ t'\ \mathit{then}\ 1\ \mathit{else}\ 0)$

The proofs of both theorems follow the standard pattern of induction followed by an exhaustive (automatic) cases analysis. The proof for *delete* requires an analogous lemma for *split_max*:

$$\mathit{split\_max}\ t = (t', a) \wedge \mathit{avl}\ t \wedge t \neq \langle\rangle \longrightarrow$$
$$\mathit{avl}\ t' \wedge h\ t = h\ t' + (\mathit{if}\ \mathit{decr}\ t\ t'\ \mathit{then}\ 1\ \mathit{else}\ 0)$$

9.5 Exercises

Exercise 9.5. We map type *'a tree_bal* back to type (*'a* × *nat*) *tree* called *'a tree_ht* in the beginning of the chapter:

$$debal :: 'a\ tree\_bal \Rightarrow ('a \times nat)\ tree$$

$$debal\ \langle\rangle = \langle\rangle$$
$$debal\ \langle l,\ (a,\ \_),\ r\rangle = \langle debal\ l,\ (a,\ max\ (h\ l)\ (h\ r) + 1),\ debal\ r\rangle$$

Prove that the AVL property is preserved: $avl\ t \longrightarrow avl\_ht\ (debal\ t)$ where avl_ht is the avl predicate on type $'a\ tree\_ht$ from the beginning of the chapter.

Define a function $debal2$ of the same type that traverses the tree only once and in particular does not use function h. Prove $avl\ t \longrightarrow debal2\ t = debal\ t$.

Exercise 9.6. If you are not into the whole *case* thing: Define a function $balL'$ by a sequence of equations where the right-hand sides do not contain *case*; you may still use $rot2$. Prove

$$(bc = Lh \longrightarrow AB \neq \langle\rangle \wedge (\nexists l\ a\ r.\ AB = \langle l,\ (a,\ Bal),\ r\rangle)) \longrightarrow$$
$$balL'\ AB\ c\ bc\ C = balL\ AB\ c\ bc\ C$$

The precondition excludes those cases for which $balL$ is underdefined.

Exercise 9.7. To realize the full space savings potential of balance factors we encode them directly into the node constructors and work with the following special tree type:

datatype $'a\ tree4 = Leaf$
  $|\ Lh\ ('a\ tree4)\ 'a\ ('a\ tree4)$
  $|\ Bal\ ('a\ tree4)\ 'a\ ('a\ tree4)$
  $|\ Rh\ ('a\ tree4)\ 'a\ ('a\ tree4)$

On this type define the AVL invariant, insertion, deletion and all necessary auxiliary functions. Prove theorems 9.6 and 9.7. Hint: modify the theory underlying Section 9.4.

10

Weight-Balanced Trees

11

Beyond Insert and Delete: ∪, ∩ and −

11.1 Specification of Union, Intersection and Difference [1]

ADT *Set2* = *Set* +

interface

$union :: 's \Rightarrow 's \Rightarrow 's$
$inter :: 's \Rightarrow 's \Rightarrow 's$
$diff :: 's \Rightarrow 's \Rightarrow 's$

specification

$invar\ s_1 \wedge invar\ s_2 \longrightarrow invar\ (union\ s_1\ s_2)$ (*union-inv*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow set\ (union\ s_1\ s_2) = set\ s_1 \cup set\ s_2$ (*union*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow invar\ (inter\ s_1\ s_2)$ (*inter-inv*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow set\ (inter\ s_1\ s_2) = set\ s_1 \cap set\ s_2$ (*inter*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow invar\ (diff\ s_1\ s_2)$ (*diff-inv*)
$invar\ s_1 \wedge invar\ s_2 \longrightarrow set\ (diff\ s_1\ s_2) = set\ s_1 - set\ s_2$ (*diff*)

Fig. 11.1. ADT *Set2*

11.2 Just Join [2]

Parameters:

[1] isabelle/src/HOL/Data_Structures/Set_Specs.thy
[2] isabelle/src/HOL/Data_Structures/Set2_Join.thy

$join$ :: $('a \times 'b)\ tree \Rightarrow 'a \Rightarrow ('a \times 'b)\ tree \Rightarrow ('a \times 'b)\ tree$
inv :: $('a \times 'b)\ tree \Rightarrow bool$

Specification:

$set\_tree\ (join\ l\ a\ r) = set\_tree\ l \cup \{a\} \cup set\_tree\ r$
$bst\ \langle l, (a, b), r\rangle \longrightarrow bst\ (join\ l\ a\ r)$
$inv\ \langle\rangle$
$inv\ l \wedge inv\ r \longrightarrow inv\ (join\ l\ a\ r)$
$inv\ \langle l, (a, b), r\rangle \longrightarrow inv\ l \wedge inv\ r$

Define union, intersection and difference via join.

```
split_min :: ('a × 'b) tree ⇒ 'a × ('a × 'b) tree
split_min ⟨l, (a, _), r⟩
= (if l = ⟨⟩ then (a, r)
    else let (m, l') = split_min l in (m, join l' a r))

join2 :: ('a × 'b) tree ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
join2 l r
= (if r = ⟨⟩ then l else let (m, r') = split_min r in join l m r')

split :: ('a × 'b) tree ⇒ 'a ⇒ ('a × 'b) tree × bool × ('a × 'b) tree
split ⟨⟩ _ = (⟨⟩, False, ⟨⟩)
split ⟨l, (a, _), r⟩ x
= (case cmp x a of
    LT ⇒ let (l₁, b, l₂) = split l x in (l₁, b, join l₂ a r) |
    EQ ⇒ (l, True, r) |
    GT ⇒ let (r₁, b, r₂) = split r x in (join l a r₁, b, r₂))
```

```
insert :: 'a ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
insert x t = (let (l, b, r) = split t x in join l x r)

delete :: 'a ⇒ ('a × 'b) tree ⇒ ('a × 'b) tree
delete x t = (let (l, b, r) = split t x in join2 l r)
```

TODO adapt slides (and base theory files?) to new defs via clauses instead of if:

```
$union :: ('a \times 'b)\ tree \Rightarrow ('a \times 'b)\ tree \Rightarrow ('a \times 'b)\ tree$
$union\ \langle\rangle\ t = t$
$union\ t\ \langle\rangle = t$
$union\ \langle l_1,\ (a,\ \_),\ r_1\rangle\ t_2$
$= (let\ (l_2,\ b_2,\ r_2) = split\ t_2\ a$
    $in\ join\ (union\ l_1\ l_2)\ a\ (union\ r_1\ r_2))$
```

```
$inter :: ('a \times 'b)\ tree \Rightarrow ('a \times 'b)\ tree \Rightarrow ('a \times 'b)\ tree$
$inter\ \langle\rangle\ t = \langle\rangle$
$inter\ t\ \langle\rangle = \langle\rangle$
$inter\ \langle l_1,\ (a,\ \_),\ r_1\rangle\ t_2$
$= (let\ (l_2,\ b_2,\ r_2) = split\ t_2\ a;$
        $l' = inter\ l_1\ l_2;\ r' = inter\ r_1\ r_2$
    $in\ if\ b_2\ then\ join\ l'\ a\ r'\ else\ join2\ l'\ r')$
```

```
$diff :: ('a \times 'b)\ tree \Rightarrow ('a \times 'b)\ tree \Rightarrow ('a \times 'b)\ tree$
$diff\ \langle\rangle\ t = \langle\rangle$
$diff\ t\ \langle\rangle = t$
$diff\ t_1\ \langle l_2,\ (a,\ \_),\ r_2\rangle$
$= (let\ (l_2,\ b_2,\ r_2) = split\ t_1\ a$
    $in\ join2\ (diff\ l_2\ l_2)\ (diff\ r_2\ r_2))$
```

11.2.1 Correctness

$split\_min\ t = (m,\ t') \wedge t \neq \langle\rangle \longrightarrow$
$m \in set\_tree\ t \wedge set\_tree\ t = \{m\} \cup set\_tree\ t'$
$split\_min\ t = (m,\ t') \wedge bst\ t \wedge t \neq \langle\rangle \longrightarrow$
$bst\ t' \wedge (\forall x{\in}set\_tree\ t'.\ m < x)$
$split\_min\ t = (m,\ t') \wedge inv\ t \wedge t \neq \langle\rangle \longrightarrow inv\ t'$

$set\_tree\ (join2\ l\ r) = set\_tree\ l \cup set\_tree\ r$
$bst\ l \wedge bst\ r \wedge (\forall x{\in}set\_tree\ l.\ \forall y{\in}set\_tree\ r.\ x < y) \longrightarrow$
$bst\ (join2\ l\ r)$
$inv\ l \wedge inv\ r \longrightarrow inv\ (join2\ l\ r)$

$split\ t\ x = (l,\ xin,\ r) \wedge bst\ t \longrightarrow$

$\mathit{set\_tree}\ l = \{a \in \mathit{set\_tree}\ t \mid a < x\} \wedge$
$\mathit{set\_tree}\ r = \{a \in \mathit{set\_tree}\ t \mid x < a\} \wedge$
$\mathit{xin} = (x \in \mathit{set\_tree}\ t) \wedge \mathit{bst}\ l \wedge \mathit{bst}\ r$
$\mathit{split}\ t\ x = (l, \mathit{xin}, r) \wedge \mathit{inv}\ t \longrightarrow \mathit{inv}\ l \wedge \mathit{inv}\ r$

$\mathit{bst}\ t \longrightarrow \mathit{set\_tree}\ (\mathit{insert}\ x\ t) = \{x\} \cup \mathit{set\_tree}\ t$
$\mathit{bst}\ t \longrightarrow \mathit{bst}\ (\mathit{insert}\ x\ t)$
$\mathit{inv}\ t \longrightarrow \mathit{inv}\ (\mathit{insert}\ x\ t)$

11.3 Join for Red-Black Trees [3]

Define the join function in Section 11.2 for red-black trees.

Function *joinL* joins two trees (and an element). Precondition: *bh l* ⩽ *bh r*. Method: Descend along the left spine of *r* until you find a subtree with the same *bheight* as *l*, then combine them into a new red node.

```
joinL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
joinL l x r
= (if bh r ⩽ bh l then R l x r
   else case r of
       ⟨l', (x', Red), r'⟩ ⇒ R (joinL l x l') x' r'
       | ⟨l', (x', Black), r'⟩ ⇒
           baliL (joinL l x l') x' r')

joinR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
joinR l x r
= (if bh l ⩽ bh r then R l x r
   else case l of
       ⟨l', (x', Red), r'⟩ ⇒ R l' x' (joinR r' x r)
       | ⟨l', (x', Black), r'⟩ ⇒
           baliR l' x' (joinR r' x r))

join :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
join l x r
= (if bh r < bh l then paint Black (joinR l x r)
   else if bh l < bh r then paint Black (joinL l x r) else B l x r)
```

[3] isabelle/src/HOL/Data_Structures/Set2_Join_RBT.thy

11.3.1 Correctness

The invariant inv for red-black trees is invc and invh, black root unnecessary for logarithmic height.

Only L, R symmetric

$$\begin{array}{l}
\textit{invc}\ l \wedge \textit{invc}\ r \wedge \textit{bh}\ l \leqslant \textit{bh}\ r \longrightarrow\\
\textit{invc2}\ (\textit{joinL}\ l\ x\ r) \wedge\\
(\textit{bh}\ l \neq \textit{bh}\ r \wedge \textit{color}\ r = \textit{Black} \longrightarrow \textit{invc}\ (\textit{joinL}\ l\ x\ r))\\
\textit{invh}\ l \wedge \textit{invh}\ r \wedge \textit{bh}\ l \leqslant \textit{bh}\ r \longrightarrow \textit{bh}\ (\textit{joinL}\ l\ x\ r) = \textit{bh}\ r\\
\textit{invh}\ l \wedge \textit{invh}\ r \wedge \textit{bh}\ l \leqslant \textit{bh}\ r \longrightarrow \textit{invh}\ (\textit{joinL}\ l\ x\ r)\\
\textit{bh}\ l \leqslant \textit{bh}\ r \longrightarrow \textit{set\_tree}\ (\textit{joinL}\ l\ x\ r) = \textit{set\_tree}\ l \cup \{x\} \cup \textit{set\_tree}\ r\\
\textit{bst}\ \langle l,\ (a,\ n),\ r\rangle \wedge \textit{bh}\ l \leqslant \textit{bh}\ r \longrightarrow \textit{bst}\ (\textit{joinL}\ l\ a\ r)
\end{array}$$

We can now prove the (trivial and) nontrivial properties of the specification just by simplification:

$$\begin{array}{l}
\textit{invc}\ l \wedge \textit{invh}\ l \wedge \textit{invc}\ r \wedge \textit{invh}\ r \longrightarrow\\
\textit{invc}\ (\textit{join}\ l\ x\ r) \wedge \textit{invh}\ (\textit{join}\ l\ x\ r)\\
\textit{set\_tree}\ (\textit{join}\ l\ x\ r) = \textit{set\_tree}\ l \cup \{x\} \cup \textit{set\_tree}\ r\\
\textit{bst}\ \langle l,\ (a,\ n),\ r\rangle \longrightarrow \textit{bst}\ (\textit{join}\ l\ a\ r)
\end{array}$$

Exercise: running time analysis

12

Arrays via Braun Trees

12.1 Arrays [1]

So far we have discussed sets (or maps) over some arbitrary linearly ordered type. Now we specialize that linearly ordered type to *nat* to model arrays. In principle we could model arrays as maps from a subset of natural numbers to the array elements. Because arrays are contiguous, it is more appropriate to model them as lists. The type $'a$ *list* comes with two array-like operations (see Appendix A):

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 $n \geqslant$ *length* xs then $xs\ !\ n$ and $xs[n := x]$ are underdefined: they are defined terms but we do not know what their value is.

Note that operationally, indexing and updating take time linear in the index, which may appear inappropriate for arrays. However, the type of lists is only an abstract model that specifies the desired functional behaviour of arrays but not their running time complexity.

The ADT of arrays is shown in Figure 12.1. Type type $'ar$ is the type of arrays, type $'a$ the type of elements in the arrays. The abstraction function *list* abstracts arrays to lists. It would make perfect sense to include *list* in the interface as well. In fact, our implementation below comes with a (reasonably efficiently) executable definition of *list*.

The behaviour of *lookup*, *update*, *size* and *array* is specified in terms of their counterparts on lists and requires that the invariant is preserved. What distinguishes the specifications of *lookup* and *update* from the standard schema (see Chapter 6) is that they carry a size precondition because the

[1] `isabelle/src/HOL/Data_Structures/Array_Specs.thy`

ADT *Array* =

interface
$lookup :: 'ar \Rightarrow nat \Rightarrow 'a$
$update :: nat \Rightarrow 'a \Rightarrow 'ar \Rightarrow 'ar$
$len :: 'ar \Rightarrow nat$
$array :: 'a\ list \Rightarrow 'ar$

abstraction $list :: 'ar \Rightarrow 'a\ list$
invariant $invar :: 'ar \Rightarrow bool$

specification
$invar\ ar \wedge n < len\ ar \longrightarrow lookup\ ar\ n = list\ ar\ !\ n$ (*lookup*)
$invar\ ar \wedge n < len\ ar \longrightarrow invar\ (update\ n\ x\ ar)$ (*update-inv*)
$invar\ ar \wedge n < len\ ar \longrightarrow list\ (update\ n\ x\ ar) = (list\ ar)[n := x]$ (*update*)
$invar\ ar \longrightarrow len\ ar = |list\ ar|$ (*len*)
$invar\ (array\ xs)$ (*array-inv*)
$list\ (array\ xs) = xs$ (*array*)

Fig. 12.1. ADT *Array*

result of *lookup* and *update* is only specified if the index is less than the size of the array.

12.2 Braun Trees [2]

One can implement arrays by any one of the many search trees presented in this book. Instead we take advantage of the fact that the keys are natural numbers and implement arrays by so-called **Braun trees** which are balanced and thus have minimal height.

The basic ideas is that we can index the nodes in a binary tree by the bit string that lead us from the root to that node. Starting from the least significant bit and while we have not reached the leading 1, we examine the bits one by one. If the current bit is 0, descend into the left subtree, otherwise into the right subtree. Instead of bit strings we use the natural numbers $\geqslant 1$ that they represent. The Braun tree with nodes indexed by 1–15 is shown in Figure 12.2. The numbers are the indexes and not the elements stored in the nodes. For example, the index 14 is 0111 in binary (least significant bit first). If you follow the path left-right-right corresponding to 011 in Figure 12.2 you reach node 14.

[2] `isabelle/src/HOL/Data_Structures/Braun_Tree.thy`

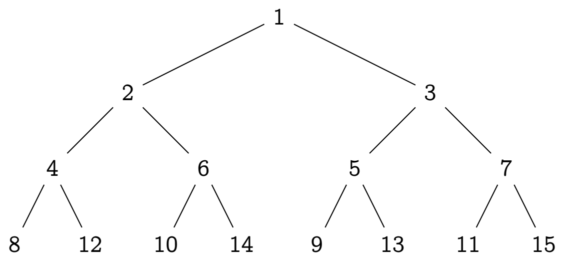
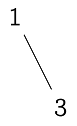


Fig. 12.2. Braun tree with nodes indexed by 1–15

A tree t is suitable for representing an array if the set of indexes of all its nodes is the interval $\{1..|t|\}$. The following tree is unsuitable because the node indexed by 2 is missing:



It turns out that the following invariant guarantees that a tree t contains exactly the nodes indexed by 1, ..., $|t|$:

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

The disjunction can alternatively be expresses as $|r| \leqslant |l| \leqslant |r| + 1$. We call a tree a **Braun tree** iff it satisfies predicate *braun*.

Although we do not need or prove this here, it is interesting to note that a tree that contains exactly the nodes indexed by 1, ..., $|t|$ is a Braun tree.

Let us now prove the earlier claim that Braun trees are balanced. First, a lemma about the composition of balanced trees:

Lemma 12.1.
$balanced\ l \wedge balanced\ r \wedge |l| = |r| + 1 \longrightarrow balanced\ \langle l,\ x,\ r\rangle$

Proof. Using Lemmas 4.7 and 4.8 and our assumptions we obtain

$$h\ \langle l,\ x,\ r\rangle = \lceil \lg\ (|r|_1 + 1)\rceil + 1 \qquad (*)$$
$$mh\ \langle l,\ x,\ r\rangle = \lfloor \lg\ |r|_1\rfloor + 1 \qquad (**)$$

Because $1 \leqslant |r|_1$ there is an i such that $2^i \leqslant |r|_1 < 2^{i+1}$ and thus $2^i < |r|_1 + 1 \leqslant 2^{i+1}$. This implies $i = \lfloor \lg\ |r|_1\rfloor$ and $i + 1 = \lceil \lg\ (|r|_1 + 1)\rceil$. Together with (*) and (**) this implies $balanced\ \langle l,\ x,\ r\rangle$. □

Now we can show that all Braun trees are balanced:

Lemma 12.2. *braun t* $\longrightarrow$ *balanced t*

Thus we know that Braun trees have optimal height (Lemma 4.6) and can even quantify it (Lemma 4.7).

Proof. The proof is by induction. We focus on the induction step where $t = \langle l,\ x,\ r\rangle$. By assumption we have *balanced l* and *balanced r*. Because of *braun t* we can distinguish two cases. First assume $|l| = |r| + 1$. The claim *balanced t* follows immediately from the previous lemma. Now assume $|l| = |r|$. By definition, there are four cases to consider when proving *balanced t*. By symmetry it suffices to consider only two of them. If *height l* $\leqslant$ *height r* and *mh r* $<$ *mh l* then *balanced t* reduces to *balanced r*, which is true by assumption. Now assume *height l* $\leqslant$ *height r* and *mh l* $\leqslant$ *mh r*. Because $|l| = |r|$, the fact that the height of a balanced tree is determined uniquely by its size (Lemma 4.7) implies *height l* $=$ *height r* and thus *balanced t* reduces to *balanced l*, which is again true by assumption. □

Note that the proof does not rely on the fact that it is the left subtree that is potentially one bigger than the right one; it merely requires that the difference in size between two siblings is at most 1.

12.3 Arrays via Braun Trees [3]

In this section we implement arrays by means of Braun trees and verify correctness and complexity. We start by defining array-like functions on Braun trees. After the above explanation of Braun trees the following lookup function will not come as a surprise:

```
lookup1 :: 'a tree ⇒ nat ⇒ 'a
lookup1 ⟨l, x, r⟩ n
= (if n = 1 then x else lookup1 (if even n then l else r) (n div 2))
```

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. This simplifies the implementation via Braun trees but is in contrast to the *Array* interface where by convention indexing starts with 0.

Function *update*1 descends in the very same manner but also performs an update when reaching 1:

[3] isabelle/src/HOL/Data_Structures/Array_Braun.thy

```
update1 :: nat ⇒ 'a ⇒ 'a tree ⇒ 'a tree
update1 _ x ⟨⟩ = ⟨⟨⟩, x, ⟨⟩⟩
update1 n x ⟨l, a, r⟩
= (if n = 1 then ⟨l, x, r⟩
   else if even n then ⟨update1 (n div 2) x l, a, r⟩
        else ⟨l, a, update1 (n div 2) x r⟩)
```

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, *update*1 $(|t| + 1)$ x t extends the tree with a new node x at index $|t| + 1$. Function *adds* iterates this process (again expecting $|t| + 1$ as the index) and thus adds a whole list of elements:

```
adds :: 'a list ⇒ nat ⇒ 'a tree ⇒ 'a tree
adds [] _ t = t
adds (x # xs) n t = adds xs (n + 1) (update1 (n + 1) x t)
```

The implementation of the *Array* interface in Figure 12.3 is just a thin wrapper around the corresponding functions on Braun trees. An array is represented as a pair of a Braun tree and its size.

```
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 ⟨⟩, |xs|)
```

Fig. 12.3. Array implementation via Braun trees

12.3.1 Functional Correctness

The invariant on arrays is obvious:

```
invar (t, l) = (braun t ∧ l = |t|)
```

The abstraction function *list* could be defined in the following intuitive way, where $[m..{<}n]$ is the list of natural numbers from m up to but excluding n (see Appendix A):

$$list\ t = map\ (lookup1\ t)\ [1..<|t| + 1]$$

Instead we define *list* recursively and derive the above equation later on

$list :: 'a\ tree \Rightarrow 'a\ list$

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

This definition is best explained by looking at Figure 12.2. The subtrees with root 2 and 3 will be mapped to the lists [2, 4, 6, 8, 10, 12, 14] and [1, 3, 5, 7, 9, 11, 13, 15]. The obvious way to combine these two lists into [1, 2, 3, ..., 15] is to splice them:

$splice :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$

$splice\ []\ ys = ys$
$splice\ (x\ \#\ xs)\ ys = x\ \#\ splice\ ys\ xs$

Note that because of this $n * \lg n$ (see Section 12.3.2) implementation of *list* we can also regard *list* as part of the interface of arrays.

Before we embark on the actual proofs we state a helpful arithmetic truth that is frequently used implicitly below:

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

where $\{m..n\} = \{k \mid m \leqslant k \wedge k \leqslant m\}$.

We will now verify that the implementation in Figure 12.3 of the *Array* interface in Figure 12.1 satisfies the given specification.

We start with property (*len*), the correctness of function *len*. Because of the invariant, (*len*) follows directly from

$$|list\ t| = |t|$$

which is proved by induction. We will also use this property implicitly in many proofs below.

The following proposition implies the correctness property (*lookup*):

$$braun\ t \wedge i < |t| \longrightarrow list\ t\ !\ i = lookup1\ t\ (i + 1) \tag{12.1}$$

The proof is by induction and uses the following proposition that is also proved by induction:

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

As a corollary to (12.1) we obtain that function *list* can indeed be expressed via *lookup*1:

$$braun\ t \longrightarrow list\ t = map\ (lookup1\ t)\ [1..<|t| + 1] \tag{12.2}$$

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. The following two preservation properties (proved by induction) 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 relating *lookup*1 and *update*1 is again proved by induction:

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

The last three properties together with (12.2) 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:

$$braun\ t \longrightarrow |update1\ (|t| + 1)\ x\ t| = |t| + 1 \tag{12.3}$$
$$braun\ t \longrightarrow braun\ (update1\ (|t| + 1)\ x\ t) \tag{12.4}$$
$$braun\ t \longrightarrow list\ (update1\ (|t| + 1)\ x\ t) = list\ t\ @\ [x] \tag{12.5}$$
$$braun\ t \longrightarrow |adds\ xs\ |t|\ t| = |t| + |xs| \wedge braun\ (adds\ xs\ |t|\ t)$$
$$braun\ t \longrightarrow list\ (adds\ xs\ |t|\ t) = list\ t\ @\ xs$$

The last two properties imply the remaining proof obligations (*array-inv*) and (*array*). The proof of (12.5) requires the following two properties of *splice* which are proved by simultaneous induction:

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

12.3.2 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 $O(n * \lg n)$.

Function *list* is similar to bottom-up merge sort and *splice* is similar to *merge*. We focus on *splice* because it performs almost all the work. Consider calling *list* on a complete tree of height h. At each level k (starting with 0 for the root) of the tree, *splice* is called 2^k times with lists of size (almost) 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 $O(2^k * 2^{h-k-1}) = O(2^{h-1}) = O(2^h)$. Thus all the splices together require time $O(h * 2^h)$. Because complete trees have size $n = 2^h$, the bound $O(n * \lg n)$ 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 $\lg 1 + \cdots + \lg n = \lg(n!)$. Now $n! \leqslant n^n$ implies $\lg(n!) \leqslant n * \lg n$. On the other hand, $n^n \leqslant (n*1)*((n-1)*2)*\cdots*(1*n) = (n!)^2$ implies $\frac{1}{2} * n * \lg n \leqslant \lg(n!)$. Thus $\lg(n!) \in \Theta(n * \lg n)$.

12.4 Flexible Arrays

Flexible arrays can be grown and shrunk at either end. Figure 12.4 shows the specification of all four operations. (For *tl* and *butlast* see Appendix A.) *Array_Flex* extends the basis specification *Array* in Figure 12.1.

ADT *Array_Flex* = *Array* +

interface
add_lo :: *'a* ⇒ *'ar* ⇒ *'ar*
del_lo :: *'ar* ⇒ *'ar*
add_hi :: *'a* ⇒ *'ar* ⇒ *'ar*
del_hi :: *'ar* ⇒ *'ar*

specification

invar ar ⟶ *invar* (*add_lo a ar*)	(*add_lo-inv*)
invar ar ⟶ *list* (*add_lo a ar*) = *a* # *list ar*	(*add_lo*)
invar ar ⟶ *invar* (*del_lo ar*)	(*del_lo-inv*)
invar ar ⟶ *list* (*del_lo ar*) = *tl* (*list ar*)	(*del_lo*)
invar ar ⟶ *invar* (*add_hi a ar*)	(*add_hi-inv*)
invar ar ⟶ *list* (*add_hi a ar*) = *list ar* @ [*a*]	(*add_hi*)
invar ar ⟶ *invar* (*del_hi ar*)	(*del_hi-inv*)
invar ar ⟶ *list* (*del_hi ar*) = *butlast* (*list ar*)	(*del_hi*)

Fig. 12.4. ADT *Array_Flex*

Below we first implement the *Array_Flex* functions on Braun trees. In a final step an implementation of *Array_Flex* on (tree,size) pairs is derived.

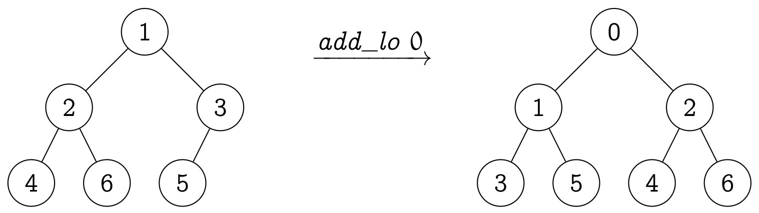
We have already seen that *update*1 adds an element at the high end. The inverse operation *del_hi* removes the high end, assuming that the given index is the size of the tree:

```
del_hi :: nat ⇒ 'a tree ⇒ 'a tree
del_hi _ ⟨⟩ = ⟨⟩
del_hi n ⟨l, x, r⟩
= (if n = 1 then ⟨⟩
   else if even n then ⟨del_hi (n div 2) l, x, r⟩
        else ⟨l, x, del_hi (n div 2) r⟩)
```

This was easy but extending an array at the low end seems hard because one has to shift the existing entries. However, Braun trees support a logarithmic implementation:

```
add_lo :: 'a ⇒ 'a tree ⇒ 'a tree
add_lo x ⟨⟩ = ⟨⟨⟩, x, ⟨⟩⟩
add_lo x ⟨l, a, r⟩ = ⟨add_lo a r, x, l⟩
```

The intended functionality is *list* $(add\_lo\ x\ t) = x\ \#\ list\ t$. Function *add_lo* installs the new element x at the root of the tree. Because *add_lo* needs to shift the indices of the elements already in the tree, the left subtree (indices 2, 4, . . .) becomes the new right subtree (indices 3, 5, . . .). The old right subtree becomes the new left subtree with the old root a added in at index 2 and the remaining elements at indices 4, 6, ... In the following example, *add_lo* 0 transforms the left tree into the right one. The numbers in the nodes are the actual elements, not their indices.



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

```
del_lo :: 'a tree ⇒ 'a tree
del_lo ⟨⟩ = ⟨⟩
del_lo ⟨l, _, r⟩ = merge l r
```

```
merge :: 'a tree ⇒ 'a tree ⇒ 'a tree
merge ⟨⟩ r = r
merge ⟨l, a, r⟩ rr = ⟨rr, a, merge l r⟩
```

Figure 12.5 shows the obvious implementation of the functions in the *Array_Flex* specification from Figure 12.4 (on the left-hand side) with the help of the corresponding Braun tree operations (on the right-hand side). It is an extension of the basic array implementation from Figure 12.3. All *Array_Flex* functions have logarithmic time complexity because the corresponding Braun tree functions do because they descend along one branch of the tree.

```
add_lo x (t, l) = (add_lo x t, l + 1)
del_lo (t, l)   = (del_lo t, l − 1)
add_hi x (t, l) = (update1 (l + 1) x t, l + 1)
del_hi (t, l)   = (del_hi l t, l − 1)
```

Fig. 12.5. Flexible array implementation via Braun trees

12.4.1 Functional Correctness

We now have to prove the properties in Figure 12.4. We have already dealt with *update*1 and thus *add_hi* above. Properties (*add_hi-inv*) and (*add_hi*) follow from (12.3), (12.4) and (12.5) stated earlier.

Correctness of *del_hi* on Braun trees is captured by the following two properties proved by induction:

$$braun\ t \longrightarrow braun\ (del\_hi\ |t|\ t)$$
$$braun\ t \longrightarrow list\ (del\_hi\ |t|\ t) = butlast\ (list\ t) \tag{12.6}$$

They imply (*del_hi-inv*) and (*del_hi*). The proof of (12.6) requires the simple fact $list\ t = [] \longleftrightarrow t = \langle\rangle$ and the following property of *splice* which is proved by induction:

$butlast\ (splice\ xs\ ys)$
$= (if\ |ys| < |xs|\ then\ splice\ (butlast\ xs)\ ys\ else\ splice\ xs\ (butlast\ ys))$

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

$braun\ t \longrightarrow braun\ (add\_lo\ x\ t)$
$braun\ t \longrightarrow list\ (add\_lo\ a\ t) = a\ \#\ list\ t$

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

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

$braun\ \langle l, x, r\rangle \longrightarrow braun\ (merge\ l\ r)$
$braun\ \langle l, x, r\rangle \longrightarrow list\ (merge\ l\ r) = splice\ (list\ l)\ (list\ r)$
$braun\ t \longrightarrow braun\ (del\_lo\ t)$
$braun\ t \longrightarrow list\ (del\_lo\ t) = tl\ (list\ t)$

The last two properties imply (*del_lo-inv*) and (*del_lo*).

12.5 Bigger, Better, Faster, More!

In this section we meet more efficient versions of some of old and new functions on Braun trees. The implementation of the corresponding array operations is trivial and is not discussed.

12.5.1 Fast Size of Braun Trees

The size of a Braun tree can be computed without having to traverse the entire tree:

```
size_fast :: 'a tree ⇒ nat
size_fast ⟨⟩ = 0
size_fast ⟨l, _, r⟩ = (let n = size_fast r in 1 + 2 · n + diff l n)

diff :: 'a tree ⇒ nat ⇒ nat
diff ⟨⟩ _ = 0
diff ⟨l, _, r⟩ n
= (if n = 0 then 1
   else if even n then diff r (n div 2 − 1) else diff l (n div 2))
```

Function *size_fast* descends down the right spine, computes the size of a *Node* as if both subtrees were the same size ($1 + 2 \cdot n$), but adds *diff l n* to compensate for bigger left subtrees. Correctness of *size_fast*

Lemma 12.3. $\mathit{braun}\ t \longrightarrow \mathit{size\_fast}\ t = |t|$

follows from this property of *diff*:

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

The running time of *size_fast* is quadratic in the height of the tree (see Exercise 12.3).

12.5.2 Initializing a Braun Tree with a Fixed Value

Above we only 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. In that case one would like to avoid the creation of the intermediate list. Of course one can call *update*1 n times, but one can also build the tree directly:

```
braun_of_naive x n
= (if n = 0 then ⟨⟩
    else let m = (n − 1) div 2
         in if odd n
            then ⟨braun_of_naive x m, x, braun_of_naive x m⟩
            else ⟨braun_of_naive x (m + 1), x,
                   braun_of_naive x m⟩)
```

This solution also has time complexity $n*\lg n$ it but it can clearly be improved by sharing identical recursive calls. Function *braun2_of* shares as much as possible by producing trees of size n and $n+1$ in parallel:

```
braun2_of :: 'a ⇒ nat ⇒ 'a tree × 'a tree
braun2_of x n
= (if n = 0 then (⟨⟩, ⟨⟨⟩, x, ⟨⟩⟩)
    else let (s, t) = braun2_of x ((n − 1) div 2)
         in if odd n then (⟨s, x, s⟩, ⟨t, x, s⟩) else (⟨t, x, s⟩, ⟨t, x, t⟩))
```

```
braun_of :: 'a ⇒ nat ⇒ 'a tree
braun_of x n = fst (braun2_of x n)
```

The running time is clearly logarithmic.

The correctness properties (see Appendix A for *replicate*)

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

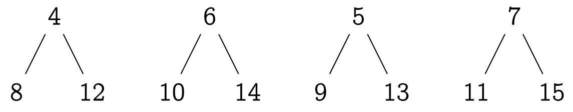
are corollaries of the more general statements

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

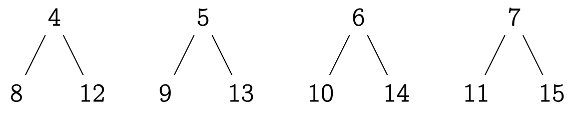
which can both be proved by induction.

12.5.3 Converting a List into a Braun Tree

We improve on function *adds* from Section 12.3 that has running time $\Theta(n * \lg n)$ 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. But we also need to get the order right. To understand how that works, consider the last two levels of the tree in Figure 12.2:



If we rearrange them in increasing order of the root labels



the following pattern emerges: the left subtrees are labeled $[8, \ldots, 11]$, the right subtrees $[12, \ldots, 15]$. Call t_i the tree with root label i. The correct order of subtrees, i.e. t_4, t_6, t_5, t_7, is restored when the three lists $[t_4,\ t_5]$, $[2,\ 3]$ and $[t_6,\ t_7]$ are combined into new trees by going through them simultaneously from left to right, yielding $[\langle t_4,\ 2,\ t_6\rangle,\ \langle t_5,\ 3,\ t_7\rangle]$, the level above.

Abstracting from this example we arrive at the following code. Loosely speaking, *brauns k xs* produces the Braun trees on level *k*.

```
brauns :: nat ⇒ 'a list ⇒ 'a tree list
brauns k xs
= (if xs = [] then []
    else let ys = take 2^k xs; zs = drop 2^k xs; ts = brauns (k + 1) zs
         in nodes ts ys (drop 2^k ts))
```

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 *take* 2^k *ts* and *drop* 2^k *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 *take* 2^k *ts* to *nodes*.

```
nodes :: 'a tree list ⇒ 'a list ⇒ 'a tree list ⇒ 'a tree list
nodes (l # ls) (x # xs) (r # rs) = ⟨l, x, r⟩ # nodes ls xs rs
nodes (l # ls) (x # xs) [] = ⟨l, x, ⟨⟩⟩ # nodes ls xs []
nodes [] (x # xs) (r # rs) = ⟨⟨⟩, x, r⟩ # nodes [] xs rs
nodes [] (x # xs) [] = ⟨⟨⟩, x, ⟨⟩⟩ # nodes [] xs []
nodes _ [] _ = []
```

Because the input list may not have exactly $2^n - 1$ elements, some of the chunks of elements and trees may be shorter than 2^k. To compensate for that, function *nodes* implicitly pads lists of trees at the end with leaves. This padding is the purpose of equations two to four.

The top-level function for turning a list into a tree simply extracts the first (and only) element from the list computed by *brauns* 0:

```
brauns1 :: 'a list ⇒ 'a tree
brauns1 xs = (if xs = [] then ⟨⟩ else brauns 0 xs ! 0)
```

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. To state this concisely we define

```
take_nths :: nat ⇒ nat ⇒ 'a list ⇒ 'a list
take_nths _ _ [] = []
take_nths i k (x # xs)
```

$= (\mathit{if}\ i = 0\ \mathit{then}\ x\ \#\ \mathit{take\_nths}\ (2^k - 1)\ k\ xs$
$\quad \mathit{else}\ \mathit{take\_nths}\ (i - 1)\ k\ xs)$

The result of *take_nths i k xs* is every 2^k-th element in *drop i xs*.

A number of simple properties follow by easy inductions:

$$\mathit{take\_nths}\ i\ k\ (\mathit{drop}\ j\ xs) = \mathit{take\_nths}\ (i + j)\ k\ xs \tag{12.7}$$

$$\mathit{take\_nths}\ 0\ 0\ xs = xs \tag{12.8}$$

$$\mathit{splice}\ (\mathit{take\_nths}\ 0\ 1\ xs)\ (\mathit{take\_nths}\ 1\ 1\ xs) = xs \tag{12.9}$$

$$\begin{aligned}&\mathit{take\_nths}\ i\ m\ (\mathit{take\_nths}\ j\ n\ xs)\\ &= \mathit{take\_nths}\ (i \cdot 2^n + j)\ (m + n)\ xs\end{aligned} \tag{12.10}$$

$$\mathit{take\_nths}\ i\ k\ xs = [] \longleftrightarrow |xs| \leqslant i \tag{12.11}$$

$$i < |xs| \longrightarrow \mathit{hd}\ (\mathit{take\_nths}\ i\ k\ xs) = xs\ !\ i \tag{12.12}$$

$$\begin{aligned}&|xs| = |ys| \lor |xs| = |ys| + 1 \longrightarrow\\ &\mathit{take\_nths}\ 0\ 1\ (\mathit{splice}\ xs\ ys) = xs\ \land\\ &\mathit{take\_nths}\ 1\ 1\ (\mathit{splice}\ xs\ ys) = ys\end{aligned} \tag{12.13}$$

$$\begin{aligned}&|\mathit{take\_nths}\ 0\ 1\ xs| = |\mathit{take\_nths}\ 1\ 1\ xs|\ \lor\\ &|\mathit{take\_nths}\ 0\ 1\ xs| = |\mathit{take\_nths}\ 1\ 1\ xs| + 1\end{aligned} \tag{12.14}$$

We also introduce a predicate relating a tree to a list:

$\mathit{braun\_list} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{list} \Rightarrow \mathit{bool}$

$\mathit{braun\_list}\ \langle\rangle\ xs = (xs = [])$
$\mathit{braun\_list}\ \langle l, x, r\rangle\ xs$
$= (xs \neq []\ \land\ x = \mathit{hd}\ xs\ \land$
$\quad \mathit{braun\_list}\ l\ (\mathit{take\_nths}\ 1\ 1\ xs)\ \land$
$\quad \mathit{braun\_list}\ r\ (\mathit{take\_nths}\ 2\ 1\ xs))$

This definition may look a bit mysterious at first but it satisfies a simple specification: $\mathit{braun\_list}\ t\ xs \longleftrightarrow \mathit{braun}\ t \land xs = \mathit{list}\ t$ (see below). The idea of the above 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*.

Lemma 12.4. $\mathit{braun\_list}\ t\ xs \longleftrightarrow \mathit{braun}\ t \land xs = \mathit{list}\ t$

Proof. The proof is by induction on *t*. The base case is trivial. In the induction step the key properties are (12.14) to prove *braun t* and (12.9) and (12.13) to prove $xs = \mathit{list}\ t$. □

The correctness proof of *brauns* rests on a few simple inductive properties:

$$|nodes\ ls\ xs\ rs| = |xs| \tag{12.15}$$

$$\begin{aligned}&i < |xs| \longrightarrow\\ &nodes\ ls\ xs\ rs\ !\ i\\ &= \langle \mathit{if}\ i < |ls|\ \mathit{then}\ ls\ !\ i\ \mathit{else}\ \langle\rangle,\ xs\ !\ i,\\ &\quad \mathit{if}\ i < |rs|\ \mathit{then}\ rs\ !\ i\ \mathit{else}\ \langle\rangle\rangle\end{aligned} \tag{12.16}$$

$$|brauns\ k\ xs| = min\ |xs|\ 2^k \tag{12.17}$$

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

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

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

where $ts = brauns\ (k + 1)\ zs$. Let $ts' = drop\ 2^k\ ts$. Below we examine $nodes\ ts\ ...\ ts'\ !\ i$ with the help of (12.16). Thus there are four similar cases of which we only discuss one representative one: assume $i < |ts| \wedge \neg\ i < |ts'|$.

$$\begin{aligned}&braun\_list\ (brauns\ k\ xs\ !\ i)\ (take\_nths\ i\ k\ xs)\\ &\longleftrightarrow braun\_list\ (nodes\ ts\ (take\ 2^k\ xs)\ ts'\ !\ i)\ (take\_nths\ i\ k\ xs)\\ &\longleftrightarrow braun\_list\ (ts\ !\ i)\ (take\_nths\ (2^k + i)\ (k + 1)\ xs) \wedge\\ &\qquad braun\_list\ \langle\rangle\ (take\_nths\ (2^{k+1} + i)\ (k + 1)\ xs)\\ &\qquad\qquad \text{by (12.16), (12.10), (12.11), (12.12) and assumptions}\\ &\longleftrightarrow True \qquad \text{by } (*)\text{, (12.11), (12.17) and assumptions}\end{aligned}$$

□

Setting $i = k = 0$ in this theorem we obtain the correctness of *brauns1* using Lemma 12.4 and (12.8):

Corollary 12.6. $braun\ (brauns1\ xs) \wedge list\ (brauns1\ xs) = xs$

Running Time Analysis

We focus on 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 $min\ n\ |xs|$. Evaluating *nodes ls xs rs* takes time linear in $|xs|$ and $|take\ n\ xs| = min\ n\ |xs|$. As a result we obtain the following definition of *t_brauns*:

```
t_brauns :: nat ⇒ 'a list ⇒ 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)
```

It is easy to prove that *t_brauns* is linear:

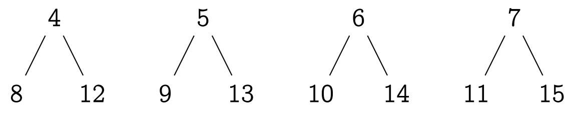
Lemma 12.7. $t\_brauns\ k\ xs = 4 \cdot |xs|$

Proof. The proof is by induction on the length of xs. If $xs = []$ the claim is trivial. Now assume $xs \neq []$ and let $zs = drop\ 2^k\ xs$.

$$\begin{aligned} &t\_brauns\ k\ xs = t\_brauns\ (k+1)\ zs + 4 \cdot min\ 2^k\ |xs| \\ &= 4 \cdot |zs| + 4 \cdot min\ 2^k\ |xs| && \text{by IH} \\ &= 4 \cdot (|xs| - 2^k) + 4 \cdot min\ 2^k\ |xs| = 4 \cdot |xs| && \square \end{aligned}$$

12.5.4 Converting a Braun Tree into a List

We improve on function *list* that has running time $O(n * \lg n)$ by developing a linear-time version. Imagine that we want to invert the computation of *brauns1* and thus of *brauns*. Thus it is natural to convert not merely a single tree but a list of trees. Looking once more at the reordered list of subtrees



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 ⇒ 'a list
list_fast_rec ts
= (let us = filter (λt. t ≠ ⟨⟩) ts
    in if us = [] then []
       else map value us @ list_fast_rec (map left us @ map right us))

value ⟨l, x, r⟩ = x
left ⟨l, x, r⟩ = l
right ⟨l, x, r⟩ = r
```

Termination of *list_fast_rec* is almost obvious because *left* and *right* remove the top node of a tree. Thus *size* seems the right measure. But if *ts* = $[\langle\rangle]$, the measure is 0 but it still leads to a recursive call (with argument $[]$). This problem can be avoided with the measure function $\varphi = \mathit{sum\_list} \circ \mathit{map}\ f$ where $f = (\lambda t.\ 2 \cdot |t| + 1)$. Assume $ts \neq []$ and let $us = \mathit{filter}\ (\lambda t.\ t \neq \langle\rangle)\ ts$. We need to show that $\varphi\ (\mathit{map}\ \mathit{left}\ us\ @\ \mathit{map}\ \mathit{right}\ us) < \varphi\ ts$. Take some t in *ts*. If $t = \langle\rangle$, $f\ t = 1$ but t is no longer in *us*, i.e. the measure decreases by 1. If $t = \langle l,\ x,\ r\rangle$ then $f\ t = 2 \cdot |l| + 2 \cdot |r| + 3$ but $f\ (\mathit{left}\ t) + f\ (\mathit{right}\ t) = 2 \cdot |l| + 2 \cdot |r| + 2$ and thus the measure also decreases by 1. Because $ts \neq []$ this shows $\varphi\ (\mathit{map}\ \mathit{left}\ us\ @\ \mathit{map}\ \mathit{right}\ us) < \varphi\ ts$. We do not show the technical details.

Finally, the top level function to extract a list from a single tree:

```
list_fast :: 'a tree ⇒ 'a list
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 indexes.

Functional Correctness

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

Theorem 12.8. $|ts| = 2^k \wedge (\forall\, i{<}2^k.\ \mathit{braun\_list}\ (ts\ !\ i)\ (\mathit{take\_nths}\ i\ k\ xs)) \longrightarrow \mathit{list\_fast\_rec}\ ts = xs$

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

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

where $n = |ts| - |xs|$. This can be proved pointwise. Take some $i < 2^k$. If $i < |xs|$ then $\mathit{take\_nths}\ i\ k\ xs = \mathit{take}\ 1\ (\mathit{drop}\ i\ xs)$ (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 $\mathit{braun\_list}\ l\ []$ and $\mathit{braun\_list}\ l\ []$ and thus $l = r = \langle\rangle$, i.e. $t\ !\ i = \langle\langle\rangle,\ xs\ !\ i,\ \langle\rangle\rangle$. If $\neg\ i < |xs|$ then $\mathit{take\_nths}\ i\ k\ xs = []$ by (12.11) and thus $\mathit{braun\_list}\ (ts\ !\ i)\ []$ by the second premise and thus $ts\ !\ i = \langle\rangle$ by definition of *braun_list*. This concludes the proof of $(*)$. The desired $\mathit{list\_fast\_rec}\ ts = xs$ follows easily by definition of *list_fast_rec*.

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

$$\begin{aligned}&\mathit{ts} \mathbin{!} i \neq \langle\rangle \wedge \mathit{root\_val}\ (\mathit{ts} \mathbin{!} i) = \mathit{xs} \mathbin{!} i \wedge\\ &\mathit{braun\_list}\ (\mathit{left}\ (\mathit{ts} \mathbin{!} i))\ (\mathit{take\_nths}\ (i + 2^k)\ (k + 1)\ \mathit{xs}) \wedge\\ &\mathit{braun\_list}\ (\mathit{right}\ (\mathit{ts} \mathbin{!} i))\ (\mathit{take\_nths}\ (i + 2 \cdot 2^k)\ (k + 1)\ \mathit{xs})\end{aligned}$$

follows from the second premise with the help of (12.10), (12.11) and (12.12). We obtain two consequences:

$$\begin{aligned}&\mathit{map\ root\_val\ ts} = \mathit{take}\ 2^k\ \mathit{xs}\\ &\mathit{list\_fast\_rec}\ (\mathit{map\ left\ ts}\ @\ \mathit{map\ right\ ts}) = \mathit{drop}\ 2^k\ \mathit{xs}\end{aligned}$$

The first consequence follows by pointwise reasoning, the second consequence with the help of the IH and (12.7). From these two consequences the desired conclusion *list_fast_rec ts* = *xs* follows by definition of *list_fast_rec*. □

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 ⇒ nat
t_list_fast_rec ts
= (let us = filter (λt. t ≠ ⟨⟩) ts
   in |ts| +
      (if us = [] then 0
       else 5 · |us| + 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.

$$\begin{aligned}&(\forall t{\in}\mathit{set\ ts}.\ t \neq \langle\rangle) \longrightarrow\\ &(\textstyle\sum t{\leftarrow}\mathit{ts}.\ k \cdot |t|)\\ &= (\textstyle\sum t{\leftarrow}\mathit{map\ left\ ts}\ @\ \mathit{map\ right\ ts}.\ k \cdot |t|) + k \cdot |\mathit{ts}|\end{aligned} \tag{12.18}$$

The suggestive notation $\sum x{\leftarrow}xs.\ f\ x$ abbreviates *sum_list* (*map f xs*).

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

Theorem 12.9. *t_list_fast_rec ts* $\leqslant (\sum t{\leftarrow}ts.\ 7 \cdot |t| + 1)$

Proof. The proof is by induction on the size of *ts*, again using the measure function $\lambda t.\ 2 \cdot |t| + 1$ which decreases with recursive calls as we proved above. If *ts* = [] the claim is trivial. Now assume *ts* ≠ [] and let *us* = *filter* $(\lambda t.\ t \neq \langle\rangle)$ *ts* and *children* = *map left us* @ *map right us*.

$$\begin{aligned}
&\mathit{t\_list\_fast\_rec}\ ts = \mathit{t\_list\_fast\_rec}\ \mathit{children} + 5 \cdot |us| + |ts| \\
&\leqslant (\textstyle\sum t \leftarrow \mathit{children}.\ 7 \cdot |t| + 1) + 5 \cdot |us| + |ts| && \text{by IH} \\
&= (\textstyle\sum t \leftarrow \mathit{children}.\ 7 \cdot |t|) + 7 \cdot |us| + |ts| \\
&= (\textstyle\sum t \leftarrow us.\ 7 \cdot |t|) + |ts| && \text{by (12.18)} \\
&\leqslant (\textstyle\sum t \leftarrow ts.\ 7 \cdot |t|) + |ts| = (\textstyle\sum t \leftarrow ts.\ 7 \cdot |t| + 1) && \square
\end{aligned}$$

12.6 Exercises

Exercise 12.1. Instead of first showing that Braun trees are balanced, give a direct proof of $\mathit{braun}\ t \longrightarrow h\ t = \lceil \lg |t|_1 \rceil$ by first showing $\mathit{braun}\ t \longrightarrow 2^{h\ t} \leqslant 2 \cdot |t| + 1$ by induction.

Exercise 12.2. One can view Braun trees as tries (see Chapter 13) by indexing them not with a *nat* but a *bool list* where each bit tells us whether to go left or right (as explained at the start of Section 12.2). Function *nat_of* specifies the intended correspondence:

$$\mathit{nat\_of} :: \mathit{bool\ list} \Rightarrow \mathit{nat}$$

$$\begin{aligned}
&\mathit{nat\_of}\ [\,] = 1 \\
&\mathit{nat\_of}\ (b \ \#\ bs) = 2 \cdot \mathit{nat\_of}\ bs + (\mathit{if}\ b\ \mathit{then}\ 1\ \mathit{else}\ 0)
\end{aligned}$$

Define the counterparts of *lookup*1 and *update*1

$$\begin{aligned}
&\mathit{lookup\_trie} :: {}'a\ \mathit{tree} \Rightarrow \mathit{bool\ list} \Rightarrow {}'a \\
&\mathit{update\_trie} :: \mathit{bool\ list} \Rightarrow {}'a \Rightarrow {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree}
\end{aligned}$$

and prove their correctness:

$$\begin{aligned}
&\mathit{braun}\ t \wedge \mathit{nat\_of}\ bs \in \{1..|t|\} \longrightarrow \\
&\mathit{lookup\_trie}\ t\ bs = \mathit{lookup}1\ t\ (\mathit{nat\_of}\ bs) \\
&\mathit{update\_trie}\ bs\ x\ t = \mathit{update}1\ (\mathit{nat\_of}\ bs)\ x\ t
\end{aligned}$$

Exercise 12.3. Function *del_lo* is defined with the help of function *merge*. Define a recursive function $\mathit{del\_lo2} :: {}'a\ \mathit{tree} \Rightarrow {}'a\ \mathit{tree}$ without recourse to any auxiliary function and prove $\mathit{del\_lo2}\ t = \mathit{del\_lo}\ t$.

Exercise 12.4. Let *lh*, the “left height”, compute the length of the left spine of a tree. Prove that the left height of a Braun tree is equal to its height: $\mathit{braun}\ t \longrightarrow \mathit{lh}\ t = h\ t$

Exercise 12.5. Show that the running time of *size_fast* is quadratic in the height of the tree: Define the running time functions *t_diff* and *t_size_fast* (taking 0 time in the base cases) and prove $\mathit{t\_size\_fast}\ t \leqslant (h\ t)^2$.

Exercise 12.6. Prove correctness of function *braun_of_naive* defined in Section 12.5.2: $\mathit{list}\ (\mathit{braun\_of\_naive}\ x\ n) = \mathit{replicate}\ n\ x$.

12.7 Bibliographic Remarks

Braun trees were investigated by Braun and Rem [27] and later, in a functional setting, by Hoogerwoord [15] who coined the term **Braun tree**. Section 12.5 is partly based on work by Okasaki [25]. The whole chapter is based on work by Nipkow and Sewell [24].

13

Tries

A trie is a search tree where keys are strings, i.e. lists. It can be viewed a tree-shaped finite automaton where the root is the start state. For example, the set of strings {a, an, can, car, cat} is encoded as the trie in Figure 13.1. The

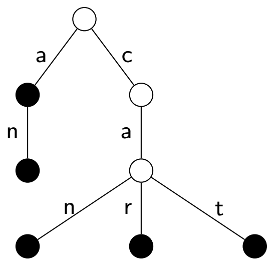


Fig. 13.1. A trie encoding {a, an, can, car, cat}

solid states are accepting, i.e. those nodes terminate the string leading to them, but they can be internal nodes.

13.1 Abstract Tries via Functions [1]

A nicely abstract model of tries is the following type:

datatype *'a trie* $=$ *Nd bool* $($*'a* $\Rightarrow$ *'a trie option*$)$

[1] isabelle/src/HOL/Data_Structures/Trie_Fun.thy

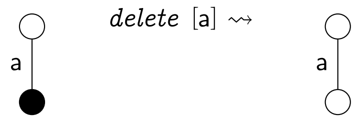
In a node *Nd b f*, *b* indicates if it is an accepting node and *f* maps characters to sub-tries. In addition to the function update notation $f(a := b)$ we use the notation

$$f(a \mapsto b) \equiv f(a := \mathit{Some}\ b)$$

This is how the ADT *Set* is implement by means of tries:

```
empty :: 'a trie
empty = Nd False (λ_. None)

isin :: 'a trie ⇒ 'a list ⇒ bool
isin (Nd b _) [] = b
isin (Nd _ m) (k # xs)
= (case m k of None ⇒ False | Some t ⇒ isin t xs)

insert :: 'a list ⇒ 'a trie ⇒ 'a trie
insert [] (Nd _ m) = Nd True m
insert (x # xs) (Nd b m)
= Nd b (m(x ↦ insert xs (case m x of None ⇒ empty | Some t ⇒ t)))

delete :: 'a list ⇒ 'a trie ⇒ 'a trie
delete [] (Nd _ m) = Nd False m
delete (x # xs) (Nd b m)
= Nd b (case m x of None ⇒ m | Some t ⇒ m(x ↦ delete xs t))
```

The definitions are straightforward. But note that *delete* does not try to shrink the trie. For example:



Formally: $\mathit{delete}\ [a]\ (\mathit{Nd\ False}\ [a \mapsto \mathit{Nd\ True}\ (\lambda\_.\ \mathit{None})]) = \mathit{Nd\ False}\ [a \mapsto \mathit{Nd\ False}\ (\lambda\_.\ \mathit{None})]$. where $[x \mapsto t] \equiv (\lambda\_.\ \mathit{None})(x \mapsto t)$. The resulting trie is correct (it represents the empty set of strings) but could have been shrunk to $\mathit{Nd\ False}\ (\lambda\_.\ \mathit{None})$.

Functional Correctness

For the correctness proof we define the following abstraction function:

```
set :: 'a trie ⇒ 'a list set
set (Nd b m)
= (if b then {[]} else {}) ∪
   (⋃ₐ case m a of None ⇒ {} | Some t ⇒ (#) a ` set t)
```

where

- $\bigcup_a f\ a$ is the union of all $f\ a$ for all a
- $f\ `\ A = \{y \mid \exists x{\in}A.\ y = f\ x\}$

The invariant is simply *True*.

The correctness properties (except for the trivial *set empty* = {}) are proved by induction:

$$isin\ t\ xs = (xs \in set\ t)$$
$$set\ (insert\ xs\ t) = set\ t \cup \{xs\}$$
$$set\ (delete\ xs\ t) = set\ t - \{xs\}$$

This simple model of tries leads to simple correctness proofs (only simple set-theoretic lemmas are needed) but is computationally inefficient because of the function space in *'a ⇒ 'a trie option*. In principle, any representation of this function space, for example by some search tree, works. However, we cannot express this in our type system. Hence we restrict ourselves to binary tries when exploring more efficient implementations of tries.

13.2 Binary Tries [2]

A **binary trie** is a trie over the alphabet *bool*. That is, binary tries represent sets of *bool list*. More concretely, every node has two sub-tries:

```
datatype trie = Lf | Nd bool (trie × trie)
```

The following auxiliary functions encode the convention that *False* selects the first and *True* the second component of a pair.

```
sel2 :: bool ⇒ 'a × 'a ⇒ 'a
sel2 b (a₁, a₂) = (if b then a₂ else a₁)
```

[2] isabelle/src/HOL/Data_Structures/Tries_Binary.thy

```
mod2 :: ('a ⇒ 'a) ⇒ bool ⇒ 'a × 'a ⇒ 'a × 'a
mod2 f b (a₁, a₂) = (if b then (a₁, f a₂) else (f a₁, a₂))
```

The implementation of the *Set* interface via binary tries is shown in Figure 13.2. Function *delete* shrinks a trie if both sub-tries have become empty.

```
empty = Lf

isin Lf ks = False
isin (Nd b lr) ks = (case ks of [] ⇒ b | k # x ⇒ isin (sel2 k lr) x)

insert [] Lf = Nd True (Lf, Lf)
insert [] (Nd b lr) = Nd True lr
insert (k # ks) Lf = Nd False (mod2 (insert ks) k (Lf, Lf))
insert (k # ks) (Nd b lr) = Nd b (mod2 (insert ks) k lr)

delete ks Lf = Lf
delete ks (Nd b lr) =
(case ks of [] ⇒ node False lr | k # ks' ⇒ node b (mod2 (delete ks') k lr))

node b lr = (if ¬ b ∧ lr = (Lf, Lf) then Lf else Nd b lr)
```

Fig. 13.2. Implementation of *Set* via binary tries

Functional Correctness

For the correctness proof we take a lazy approach and define the abstraction function in a trivial manner via *isin*:

$$\mathit{set\_trie} :: \mathit{trie} \Rightarrow \mathit{bool\ list\ set}$$
$$\mathit{set\_trie}\ t = \{xs \mid \mathit{isin}\ t\ xs\}$$

Defining the abstraction function via one of the interface functions is not advisable if there is a higher-level mathematical definition that simplifies the proofs. This was the case in the correctness proof for tries based on functions in Section 13.1 but is not the case here.

We are also lazy in that we set the invariant to *True*. A more precise invariant would express that the tries are minimal, i.e. cannot be shrunk. See Exercise 13.2. It turns out that the correctness properties do not require this more precise invariant.

The required correctness properties (ignoring the trivial *empty*) are

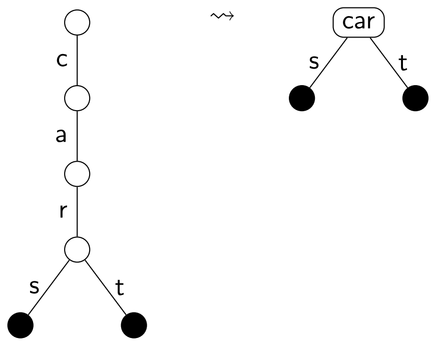
$$isin\ t\ xs = (xs \in set\_trie\ t) \tag{13.1}$$
$$set\_trie\ (insert\ xs\ t) = set\_trie\ t \cup \{xs\} \tag{13.2}$$
$$set\_trie\ (delete\ xs\ t) = set\_trie\ t - \{xs\} \tag{13.3}$$

The one for *isin* holds trivially because of our definition of *set_trie*. The ones for *insert* and *delete* are simple consequences of the following inductive properties:

$$isin\ (insert\ xs\ t)\ ys = (xs = ys \lor isin\ t\ ys)$$
$$isin\ (delete\ xs\ t)\ ys = (xs \neq ys \land isin\ t\ ys)$$

13.3 Binary Patricia Tries [3]

Tries can contain long branches without branching. These can be contracted by storing the branch directly in the start node. The result is called a **Patricia trie**. The following figure shows the contraction of a trie into a Patricia trie:



This is the datatype of (binary) Patricia tries:

datatype $trieP = LfP \mid NdP\ (bool\ list)\ bool\ (trieP \times trieP)$

The implementation of the *Set* interface via binary Patricia tries is shown in Figure 13.3.

Functional Correctness

The correctness proof does not directly show that *trieP* implements the *Set* ADT. Instead we take advantage of the fact that we have already proved that

[3] isabelle/src/HOL/Data_Structures/Tries_Binary.thy

```
emptyP = LfP

isinP LfP ks = False
isinP (NdP ps b lr) ks =
(let n = |ps|
 in if ps = take n ks
    then case drop n ks of [] ⇒ b | k # x ⇒ isinP (sel2 k lr) x else False)

insertP ks LfP = NdP ks True (LfP, LfP)
insertP ks (NdP ps b lr) =
(case split ks ps of (qs, [], []) ⇒ NdP ps True lr
 | (qs, [], p # ps') ⇒
     let t = NdP ps' b lr in NdP qs True (if p then (LfP, t) else (t, LfP))
 | (qs, k # ks', []) ⇒ NdP ps b (mod2 (insertP ks') k lr)
 | (qs, k # ks', p # ps') ⇒
     let tp = NdP ps' b lr; tk = NdP ks' True (LfP, LfP)
     in NdP qs False (if k then (tp, tk) else (tk, tp)))

deleteP ks LfP = LfP
deleteP ks (NdP ps b lr) =
(case split ks ps of
   (qs, ks', p # ps') ⇒ NdP ps b lr
 | (qs, k # ks', []) ⇒ nodeP ps b (mod2 (deleteP ks') k lr)
 | (qs, [], []) ⇒ nodeP ps False lr)

nodeP ps b lr = (if ¬ b ∧ lr = (LfP, LfP) then LfP else NdP ps b lr)
```

Fig. 13.3. Implementation of *Set* via binary Patricia tries

trie implements the *Set* ADT and merley relate *trieP* back to *trie* via an abstraction function. This is an exercise in stepwise data refinement.

The abstraction function *abs_trieP* is defined via an auxiliary function that prefixes a trie with a bit list:

```
prefix_trie :: bool list ⇒ trie ⇒ trie
prefix_trie [] t = t
prefix_trie (k # ks) t =
(let t' = prefix_trie ks t in Nd False (if k then (Lf, t') else (t', Lf)))

abs_trieP :: trieP ⇒ trie
abs_trieP LfP = Lf
abs_trieP (NdP ps b (l, r)) =
prefix_trie ps (Nd b (abs_trieP l, abs_trieP r))
```

Again we take a lazy approach and set the invariant on *trieP* to *True*.

Correctness of *emptyP* is trivial. Correctness of the remaining operations is proved by induction and requires a number of supporting inductive lemmas which we display before the corresponding correctness property.

Correctness of *isinP*:

$$isin\ (prefix\_trie\ ps\ t)\ ks = (ps = take\ |ps|\ ks \wedge isin\ t\ (drop\ |ps|\ ks))$$

$$isinP\ t\ ks = isin\ (abs\_trieP\ t)\ ks \tag{13.4}$$

Correctness of *insertP*:

$$prefix\_trie\ ks\ (Nd\ True\ (Lf,\ Lf)) = insert\ ks\ Lf$$
$$insert\ ps\ (prefix\_trie\ ps\ (Nd\ b\ lr)) = prefix\_trie\ ps\ (Nd\ True\ lr)$$
$$insert\ (ks\ @\ ks')\ (prefix\_trie\ ks\ t) = prefix\_trie\ ks\ (insert\ ks'\ t)$$
$$prefix\_trie\ (ps\ @\ qs)\ t = prefix\_trie\ ps\ (prefix\_trie\ qs\ t)$$
$$split\ ks\ ps = (qs,\ ks',\ ps') \longrightarrow$$
$$ks = qs\ @\ ks' \wedge ps = qs\ @\ ps' \wedge$$
$$(ks' \neq [] \wedge ps' \neq [] \longrightarrow hd\ ks' \neq hd\ ps')$$

$$abs\_trieP\ (insertP\ ks\ t) = insert\ ks\ (abs\_trieP\ t) \tag{13.5}$$

Correctness of *deleteP*:

$$(prefix\_trie\ xs\ t = Lf) = (xs = [] \wedge t = Lf)$$
$$(abs\_trieP\ t = Lf) = (t = LfP)$$
$$delete\ xs\ (prefix\_trie\ xs\ (Nd\ b\ (l,\ r))) =$$
$$(if\ (l,\ r) = (Lf,\ Lf)\ then\ Lf\ else\ prefix\_trie\ xs\ (Nd\ False\ (l,\ r)))$$
$$delete\ (xs\ @\ ys)\ (prefix\_trie\ xs\ t) =$$
$$(if\ delete\ ys\ t = Lf\ then\ Lf\ else\ prefix\_trie\ xs\ (delete\ ys\ t))$$

$$abs\_trieP\ (deleteP\ ks\ t) = delete\ ks\ (abs\_trieP\ t) \tag{13.6}$$

It is now trivial to obtain the correctness of the *trieP* implementation of sets. The abstraction function is simply the composition of the two abstraction abstraction functions: $set\_trieP = set\_trie \circ abs\_trieP$. The required correctness properties

$$isinP\ t\ xs = (xs \in set\_trieP\ t)$$
$$set\_trieP\ (insertP\ xs\ t) = set\_trieP\ t \cup \{xs\}$$
$$set\_trieP\ (deleteP\ xs\ t) = set\_trieP\ t - \{xs\}$$

are trivial compositions of the correctness properties (13.1)–(13.3) and (13.4)–(13.6).

13.4 Exercises

Exercise 13.1. Rework the above theory of binary tries as follows. Eliminate the *bool* argument from constructor *Nd* by replacing *Nd* by two constructors representing *Nd True* and *Nd False*.

Exercise 13.2. Define the invariant *invar* that characterizes fully shrunk binary tries, i.e. tries where every non-*Lf* sub-trie represents a non-empty set. Note that a trie represents the empty set if it does not contain any node *Nd True* _. Prove that *insert* and *delete* maintain the invariant.

Exercise 13.3. Repeat the previous exercise for *trieP*. TODO

13.5 Bibliographic Remarks

Tries were first sketched by De La Briandais [7] and described in more detail by Fredkin [11] who coined their name based on the word reTRIEval. However, “trie” is usually pronounced like “try” rather than “tree” to avoid confusion. Patricia tries are due to Morrison [21].

Part III

More Trees

14

Finger Trees (Peter Lammich)

15

Quad Trees

16

Huffman's Algorithm [1]

Huffman's algorithm [16] is a simple and elegant procedure for constructing a binary tree with minimum weighted path length—a measure of cost that considers both the lengths of the paths from the root to the leaf nodes and the weights associated with the leaf nodes. The algorithm's main application is data compression: By equating leaf nodes with characters, and weights with character frequencies, we can use it to derive optimum binary codes. A *binary code* is a map from characters to nonempty sequences.

This chapter presents Huffman's algorithm and its optimality proof. The proof follows Knuth's informal argument [19]. This chapter's text is based on a published article [5]. An alternative formal proof, in Coq, is due to Théry [28].

16.1 Binary Codes

Suppose we want to encode strings over a finite source alphabet as sequences of bits. Fixed-length codes like ASCII are simple and fast, but they generally waste space. If we know the frequency w_a of each source symbol a, we can save space by using shorter code words for the most frequent symbols. We say that a variable-length code is *optimum* if it minimizes the sum $\sum_a w_a\,\delta_a$, where δ_a is the length of the binary code word for a.

As an example, consider the string '*abacabad*'. Encoding it with the code

$$\mathsf{C}_1 = \{a \mapsto 0,\ b \mapsto 10,\ c \mapsto 110,\ d \mapsto 111\}$$

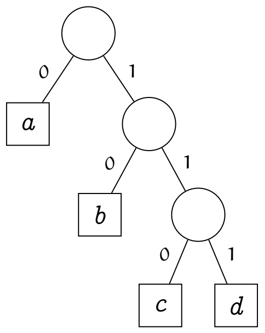
gives the 14-bit code word 01001100100111. The code C_1 is optimum: No code that unambiguously encodes source symbols one at a time could do better than C_1 on the input '*abacabad*'. With a fixed-length code such as

[1] `AFP/Huffman/Huffman.thy`

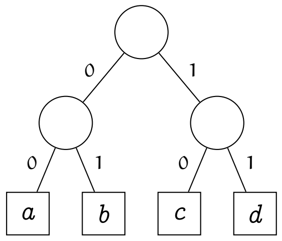
$$C_2 = \{a \mapsto 00,\ b \mapsto 01,\ c \mapsto 10,\ d \mapsto 11\}$$

we need at least 16 bits to encode the same string.

Binary codes can be represented by binary trees. For example, the trees



and



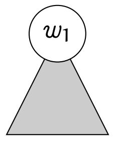
correspond to C_1 and C_2. The code word for a given symbol can be obtained as follows: Start at the root and descend toward the leaf node associated with the symbol one node at a time; emit a 0 whenever the left child of the current node is chosen and a 1 whenever the right child is chosen. The generated sequence of 0s and 1s is the code word.

To avoid ambiguities, we require that only leaf nodes are labeled with symbols. This ensures that no code word is a prefix of another. Moreover, it is sufficient to consider only full binary trees (trees whose inner nodes all have two children), because any node with only one child can advantageously be eliminated by removing it and letting the child take its parent's place.

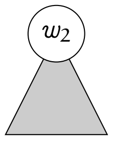
Each node in a code tree is assigned a *weight*. For a leaf node, the weight is the frequency of its symbol; for an inner node, it is the sum of the weights of its subtrees. In diagrams, we often annotate the nodes with their weights.

16.2 The Algorithm

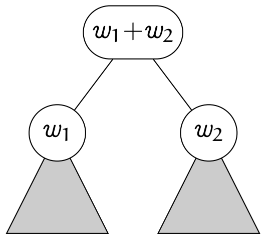
David Huffman [16] discovered a simple algorithm for constructing an optimum code tree for specified symbol frequencies: Create a forest consisting of only leaf nodes, one for each symbol in the alphabet, taking the given symbol frequencies as initial weights for the nodes. Then pick the two trees



and

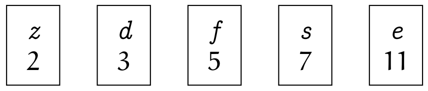


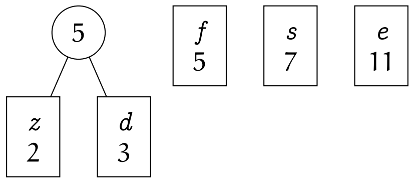
with the lowest weights and replace them with the tree

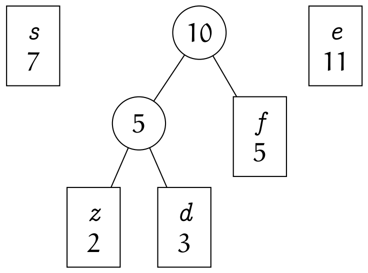


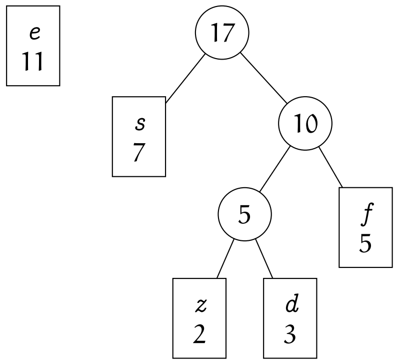
Repeat this process until only one tree is left.

As an illustration, executing the algorithm for the frequencies $f_d = 3$, $f_e = 11$, $f_f = 5$, $f_s = 7$, $f_z = 2$ gives rise to the following sequence of states:

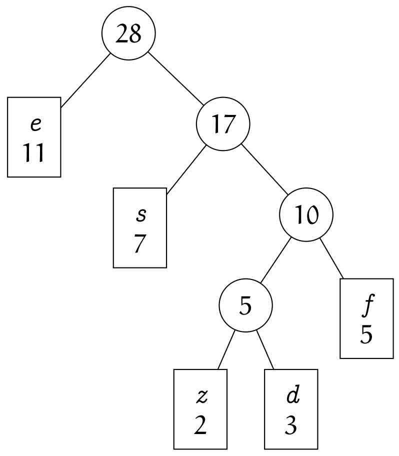
1.


2.


3.


4.


5.



The resulting tree is optimum for the given frequencies.

16.3 The Implementation

The functional implementation of the algorithm relies on the following type:

datatype *'a tree* = *Leaf nat 'a* | *Node nat* (*'a tree*) (*'a tree*)

Leaf nodes are of the form *Leaf* w a, where a is a symbol and w is the frequency associated with a, and inner nodes are of the form *Node* w t_1 t_2, where t_1 and t_2 are the left and right subtrees and w caches the sum of the weights of t_1 and t_2. The *cachedWeight* function extracts the weight stored in a node:

cachedWeight (*Leaf* w a) = w
cachedWeight (*Node* w t_1 t_2) = w

The implementation builds on two additional auxiliary functions. The first one, *uniteTrees*, combines two trees by adding an inner node above them:

uniteTrees t_1 t_2 = *Node* (*cachedWeight* t_1 + *cachedWeight* t_2) t_1 t_2

The second function, *insortTree*, inserts a tree into a forest sorted by cached weight, preserving the sort order:

$$\begin{array}{l} \mathit{insortTree}\ u\ []\ =\ [u] \\ \mathit{insortTree}\ u\ (t\ \#\ ts) \\ =\ (\mathit{if}\ \mathit{cachedWeight}\ u\ \leqslant\ \mathit{cachedWeight}\ t\ \mathit{then}\ u\ \#\ t\ \#\ ts \\ \quad \mathit{else}\ t\ \#\ \mathit{insortTree}\ u\ ts) \end{array}$$

The main function that implements Huffman's algorithm follows:

$$\begin{array}{l} \mathit{huffman}\ [t]\ =\ t \\ \mathit{huffman}\ (t_1\ \#\ t_2\ \#\ ts)\ =\ \mathit{huffman}\ (\mathit{insortTree}\ (\mathit{uniteTrees}\ t_1\ t_2)\ ts) \end{array}$$

The function should initially be invoked with a nonempty list of leaf nodes sorted by weight. It repeatedly unites the first two trees of the forest it receives as argument until a single tree is left.

16.4 Basic Auxiliary Functions Needed for the Proof

This section introduces basic concepts such as alphabet, consistency, and optimality, which are needed to state the correctness and optimality of Huffman's algorithm. The next section introduces more specialized functions that arise in the proof.

The *alphabet* of a code tree is the set of symbols appearing in the tree's leaf nodes:

$$\begin{array}{l} \mathit{alphabet}\ (\mathit{Leaf}\ w\ a)\ =\ \{a\} \\ \mathit{alphabet}\ (\mathit{Node}\ w\ t_1\ t_2)\ =\ \mathit{alphabet}\ t_1\ \cup\ \mathit{alphabet}\ t_2 \end{array}$$

A tree is *consistent* if for each inner node the alphabets of the two subtrees are disjoint. Intuitively, this means that a symbol occurs in at most one leaf node. Consistency is a sufficient condition for δ_a (the length of the code word for a) to be uniquely defined. This well-formedness property appears as an assumption in many of the Isabelle lemmas. The definition follows:

$$\begin{array}{l} \mathit{consistent}\ (\mathit{Leaf}\ w\ a)\ =\ \mathit{True} \\ \mathit{consistent}\ (\mathit{Node}\ w\ t_1\ t_2) \\ =\ (\mathit{consistent}\ t_1\ \wedge\ \mathit{consistent}\ t_2\ \wedge\ \mathit{alphabet}\ t_1\ \cap\ \mathit{alphabet}\ t_2\ =\ \{\}) \end{array}$$

The *depth* of a symbol (which we wrote as δ_a above) is the length of the path from the root to that symbol, or equivalently the length of the code word for the symbol:

$$
\begin{aligned}
&\mathit{depth}\ (\mathit{Leaf}\ w\ b)\ a = 0\\
&\mathit{depth}\ (\mathit{Node}\ w\ t_1\ t_2)\ a\\
&= (\mathit{if}\ a \in \mathit{alphabet}\ t_1\ \mathit{then}\ \mathit{depth}\ t_1\ a + 1\\
&\quad \mathit{else\ if}\ a \in \mathit{alphabet}\ t_2\ \mathit{then}\ \mathit{depth}\ t_2\ a + 1\ \mathit{else}\ 0)
\end{aligned}
$$

By convention, symbols that do not occur in the tree or that occur at the root of a one-node tree are given a depth of 0. If a symbol occurs in several leaf nodes (of an inconsistent tree), the depth is arbitrarily defined in terms of the leftmost node labeled with that symbol.

The *height* of a tree is the length of the longest path from the root to a leaf node, or equivalently the length of the longest code word:

$$
\begin{aligned}
&\mathit{height}\ (\mathit{Leaf}\ w\ a) = 0\\
&\mathit{height}\ (\mathit{Node}\ w\ t_1\ t_2) = \mathit{max}\ (\mathit{height}\ t_1)\ (\mathit{height}\ t_2) + 1
\end{aligned}
$$

The *frequency* of a symbol (which we wrote as w_a above) is the sum of the weights attached to the leaf nodes labeled with that symbol:

$$
\begin{aligned}
&\mathit{freq}\ (\mathit{Leaf}\ w\ a)\ b = (\mathit{if}\ b = a\ \mathit{then}\ w\ \mathit{else}\ 0)\\
&\mathit{freq}\ (\mathit{Node}\ w\ t_1\ t_2)\ b = \mathit{freq}\ t_1\ b + \mathit{freq}\ t_2\ b
\end{aligned}
$$

For consistent trees, the sum comprises at most one nonzero term. The frequency is then the weight of the leaf node labeled with the symbol, or 0 if there is no such node.

Two trees are *comparable* if they have the same alphabet and symbol frequencies. This is an important concept, because it allows us to state not only that the tree constructed by Huffman's algorithm is optimal but also that it has the expected alphabet and frequencies.

The *weight* function returns the weight of a tree:

$$
\begin{aligned}
&\mathit{weight}\ (\mathit{Leaf}\ w\ a) = w\\
&\mathit{weight}\ (\mathit{Node}\ w\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{weight}\ t_2
\end{aligned}
$$

In the *Node* case, we ignore the weight cached in the node and instead compute the tree's weight recursively.

The *cost* (or *weighted path length*) of a consistent tree is the sum $\sum_{a \in \mathit{alphabet}\ t} \mathit{freq}\ t\ a \cdot \mathit{depth}\ t\ a$ (which we wrote as $\sum_a w_a \delta_a$ above). It is defined recursively by

$$\begin{aligned}
&\mathit{cost}\ (\mathit{Leaf}\ w\ a) = 0\\
&\mathit{cost}\ (\mathit{Node}\ w\ t_1\ t_2) = \mathit{weight}\ t_1 + \mathit{cost}\ t_1 + \mathit{weight}\ t_2 + \mathit{cost}\ t_2
\end{aligned}$$

A tree is optimum if and only if its cost is not greater than that of any comparable tree:

$$\begin{aligned}
&\mathit{optimum}\ t\\
&= (\forall u.\ \mathit{consistent}\ u \longrightarrow\\
&\qquad \mathit{alphabet}\ t = \mathit{alphabet}\ u \longrightarrow\\
&\qquad \mathit{freq}\ t = \mathit{freq}\ u \longrightarrow \mathit{cost}\ t \leqslant \mathit{cost}\ u)
\end{aligned}$$

Tree functions are readily generalized to forests; for example, the alphabet of a forest is defined as the union of the alphabets of its trees. The forest generalizations have a subscript ‘F’ attached to their name (e.g. $\mathit{alphabet}_F$).

16.5 Other Functions Needed for the Proof

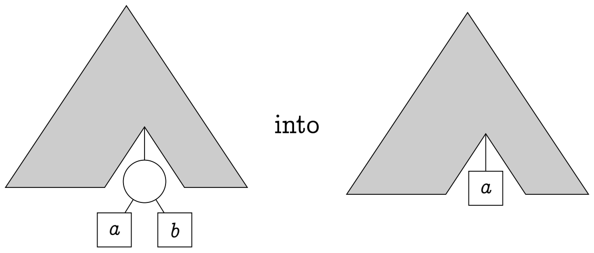
The optimality proof needs to interchange nodes in trees, to replace a two-leaf subtree with weights w_1 and w_2 by a single leaf of weight $w_1 + w_2$ and vice versa, and to refer to the two symbols with the lowest frequencies. These concepts are represented by four functions: *swapFourSyms*, *mergeSibling*, *splitLeaf*, and *minima*.

The four-way symbol interchange function *swapFourSyms* takes four symbols a, b, c, d with $a \neq b$ and $c \neq d$, and exchanges them so that a and b occupy c's and d's positions. A naive definition of this function would be $\mathit{swapSyms}\ (\mathit{swapSyms}\ t\ a\ c)\ b\ d$, where *swapSyms* exchanges two symbols. This naive definition fails in the face of aliasing: If $a = d$, but $b \neq c$, then $\mathit{swapFourSyms}\ a\ b\ c\ d$ would leave a in b's position.

The following lemma about *swapSyms* captures the intuition that in order to minimize the cost, more frequent symbols should be encoded using fewer bits than less frequent ones:

Lemma 16.1. $\mathit{consistent}\ t \land a \in \mathit{alphabet}\ t \land b \in \mathit{alphabet}\ t \land \mathit{freq}\ t\ a \leqslant \mathit{freq}\ t\ b \land \mathit{depth}\ t\ a \leqslant \mathit{depth}\ t\ b \longrightarrow \mathit{cost}\ (\mathit{swapSyms}\ t\ a\ b) \leqslant \mathit{cost}\ t$

Given a symbol a, the *mergeSibling* function transforms the tree



The frequency of a in the resulting tree is the sum of the original frequencies of a and b. The function is defined by the equations

$$
\begin{array}{l}
\mathit{mergeSibling}\ (\mathit{Leaf}\ w_b\ b)\ a = \mathit{Leaf}\ w_b\ b \\
\mathit{mergeSibling}\ (\mathit{Node}\ w\ (\mathit{Leaf}\ w_b\ b)\ (\mathit{Leaf}\ w_c\ c))\ a \\
= (\mathit{if}\ a = b \lor a = c\ \mathit{then}\ \mathit{Leaf}\ (w_b + w_c)\ a \\
\quad \mathit{else}\ \mathit{Node}\ w\ (\mathit{Leaf}\ w_b\ b)\ (\mathit{Leaf}\ w_c\ c)) \\
\mathit{mergeSibling}\ (\mathit{Node}\ w\ t_1\ t_2)\ a \\
= \mathit{Node}\ w\ (\mathit{mergeSibling}\ t_1\ a)\ (\mathit{mergeSibling}\ t_2\ a)
\end{array}
$$

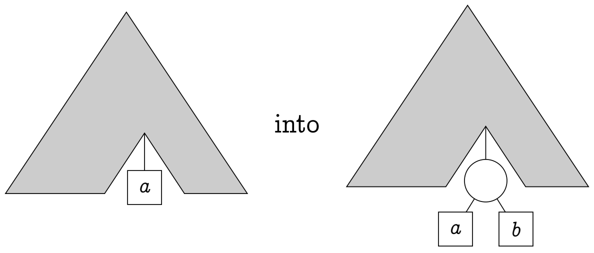
The defining equations are applied sequentially: The third equation is applicable only if the second one does not match.

The *sibling* function returns the label of the node that is the (left or right) sibling of the node labeled with the given symbol a in tree t. If a is not in t's alphabet or it occurs in a node with no sibling leaf, we simply return a. This gives us the nice property that if t is consistent, then $\mathit{sibling}\ t\ a \neq a$ if and only if a has a sibling. The definition, which is omitted here, distinguishes the same cases as *mergeSibling*.

Using the *sibling* function, we can state that merging two sibling leaves with weights w_a and w_b decreases the cost by $w_a + w_b$:

Lemma 16.2. $\mathit{consistent}\ t \land \mathit{sibling}\ t\ a \neq a \longrightarrow \mathit{cost}\ (\mathit{mergeSibling}\ t\ a) + \mathit{freq}\ t\ a + \mathit{freq}\ t\ (\mathit{sibling}\ t\ a) = \mathit{cost}\ t$

The *splitLeaf* function undoes the merging performed by *mergeSibling*: Given two symbols a, b and two frequencies w_a, w_b, it transforms



In the resulting tree, a has frequency w_a and b has frequency w_b. We normally invoke *splitLeaf* with w_a and w_b such that $freq\ t\ a = w_a + w_b$. The definition follows:

$$\begin{aligned}
&splitLeaf\ (Leaf\ w_c\ c)\ w_a\ a\ w_b\ b\\
&= (if\ c = a\ then\ Node\ w_c\ (Leaf\ w_a\ a)\ (Leaf\ w_b\ b)\ else\ Leaf\ w_c\ c)\\
&splitLeaf\ (Node\ w\ t_1\ t_2)\ w_a\ a\ w_b\ b\\
&= Node\ w\ (splitLeaf\ t_1\ w_a\ a\ w_b\ b)\ (splitLeaf\ t_2\ w_a\ a\ w_b\ b)
\end{aligned}$$

Splitting a leaf with weight $w_a + w_b$ into two sibling leaves with weights w_a and w_b increases the cost by $w_a + w_b$:

Lemma 16.3. $consistent\ t \wedge a \in alphabet\ t \wedge freq\ t\ a = w_a + w_b \longrightarrow cost\ (splitLeaf\ t\ w_a\ a\ w_b\ b) = cost\ t + w_a + w_b$

Finally, the *minima* predicate expresses that two symbols a, b have the lowest frequencies in the tree t and that $freq\ t\ a \leqslant freq\ t\ b$:

$$\begin{aligned}
&minima\ t\ a\ b\\
&= (a \in alphabet\ t \wedge b \in alphabet\ t \wedge a \neq b \wedge\\
&\quad (\forall\, c \in alphabet\ t.\\
&\qquad c \neq a \longrightarrow c \neq b \longrightarrow freq\ t\ a \leqslant freq\ t\ c \wedge freq\ t\ b \leqslant freq\ t\ c))
\end{aligned}$$

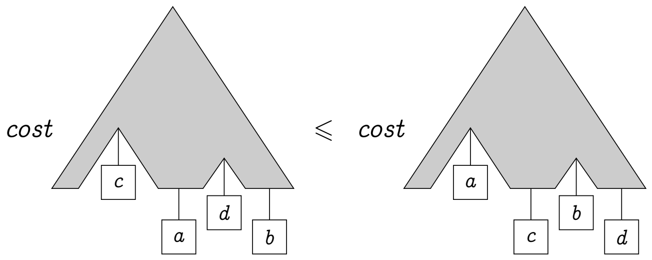
16.6 The Key Lemmas and Theorems

It is easy to prove that the tree returned by Huffman's algorithm preserves the alphabet, consistency, and symbol frequencies of the original forest:

$$\begin{aligned}
&ts \neq [] \longrightarrow alphabet\ (huffman\ ts) = alphabet_F\ ts\\
&consistent_F\ ts \wedge ts \neq [] \longrightarrow consistent\ (huffman\ ts)\\
&ts \neq [] \longrightarrow freq\ (huffman\ ts)\ a = freq_F\ ts\ a
\end{aligned}$$

The main difficulty is to prove the optimality of the tree constructed by Huffman's algorithm. We need three lemmas before we can present the optimality theorem.

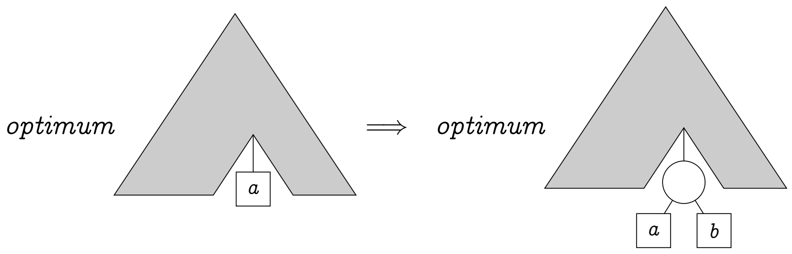
First, if a and b are minima, and c and d are at the very bottom of the tree, then exchanging a and b with c and d does not increase the tree's cost. Graphically, we have



Lemma 16.4. *consistent* t $\wedge$ *minima* t a b $\wedge$ $c \in$ *alphabet* t $\wedge$ $d \in$ *alphabet* t $\wedge$ *depth* t c $=$ *height* t $\wedge$ *depth* t d $=$ *height* t $\wedge$ $c \neq d \longrightarrow$ *cost* (*swapFourSyms* t a b c d) $\leqslant$ *cost* t

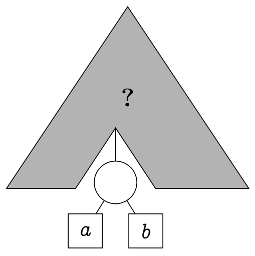
Proof. The proof is by case distinctions on $a = c$, $a = d$, $b = c$, and $b = d$. The cases are easy to prove by expanding the definition of *swapFourSyms* and applying Lemma 16.1. □

The tree *splitLeaf* t w_a a w_b b is optimum if t is optimum, under a few assumptions, notably that a and b are minima of the new tree and that *freq* t $a = w_a + w_b$. Graphically:



Lemma 16.5. *consistent* t $\wedge$ *optimum* t $\wedge$ $a \in$ *alphabet* t $\wedge$ $b \notin$ *alphabet* t $\wedge$ *freq* t a $=$ $w_a + w_b$ $\wedge$ ($\forall$ $c \in$*alphabet* t. $w_a \leqslant$ *freq* t c $\wedge$ $w_b \leqslant$ *freq* t c) $\longrightarrow$ *optimum* (*splitLeaf* t w_a a w_b b)

Proof. We assume that t's cost is less than or equal to that of any other comparable tree v and show that *splitLeaf* t w_a a w_b b has a cost less than or equal to that of any other comparable tree u. For the nontrivial case where *height* $t > 0$, it is easy to prove that there must be two symbols c and d occurring in sibling nodes at the very bottom of u. From u we construct the tree *swapFourSyms* u a b c d in which the minima a and b are siblings:



The question mark reminds us that we know nothing specific about u's structure. Merging a and b gives a tree comparable with t, which we can use to instantiate v:

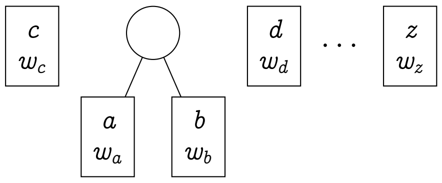
$$\begin{aligned}
&\mathit{cost}\ (\mathit{splitLeaf}\ t\ a\ w_a\ b\ w_b) = \mathit{cost}\ t + w_a + w_b && \text{by Lemma 16.3}\\
&\leqslant \mathit{cost}\ (\mathit{mergeSibling}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d)\ a) + w_a + w_b &&\\
& && \text{by optimality assumption}\\
&= \mathit{cost}\ (\mathit{swapFourSyms}\ u\ a\ b\ c\ d) && \text{by Lemma 16.2}\\
&\leqslant \mathit{cost}\ u && \text{by Lemma 16.4} \quad \square
\end{aligned}$$

An important property of Huffman's algorithm is that once it has combined two lowest-weight trees using *uniteTrees*, it does not visit these trees ever again. This suggests that splitting a leaf node before applying the algorithm should give the same result as applying the algorithm first and splitting the leaf node afterward.

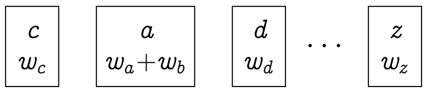
Lemma 16.6. $\mathit{consistent}_F\ ts \wedge ts \neq [] \wedge a \in \mathit{alphabet}_F\ ts \wedge \mathit{freq}_F\ ts\ a = w_a + w_b \longrightarrow \mathit{splitLeaf}\ (\mathit{huffman}\ ts)\ w_a\ a\ w_b\ b = \mathit{huffman}\ (\mathit{splitLeaf}_F\ ts\ w_a\ a\ w_b\ b)$

The proof is by straightforward induction on the length of the forest ts.

As a consequence of this commutativity lemma, applying Huffman's algorithm on a forest of the form



gives the same result as applying the algorithm on the "flat" forest



followed by splitting the leaf node a into two nodes a and b with frequencies w_a, w_b. The lemma effectively provides a way to flatten the forest at each step of the algorithm.

This leads us to our main result.

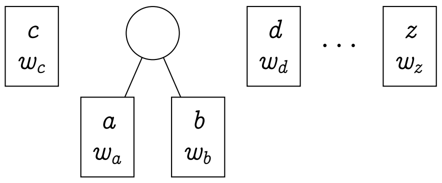
Theorem 16.7. $\mathit{consistent}_F\ ts \wedge \mathit{height}_F\ ts = 0 \wedge \mathit{sortedByWeight}\ ts \wedge ts \neq [] \longrightarrow \mathit{optimum}\ (\mathit{huffman}\ ts)$

Proof. The proof is by induction on the length of ts. The assumptions ensure that ts is of the form

$$\boxed{\begin{array}{c}a\\w_a\end{array}}\quad\boxed{\begin{array}{c}b\\w_b\end{array}}\quad\boxed{\begin{array}{c}c\\w_c\end{array}}\quad\boxed{\begin{array}{c}d\\w_d\end{array}}\quad\cdots\quad\boxed{\begin{array}{c}z\\w_z\end{array}}$$

with $w_a \leqslant w_b \leqslant w_c \leqslant w_d \leqslant \cdots \leqslant w_z$. If *ts* consists of a single node, the node has cost 0 and is therefore optimum. If *ts* has length 2 or more, the first step of the algorithm leaves us with a term such as

huffman



In the diagram, we put the newly created tree at position 2 in the forest; in general, it could be anywhere. By Lemma 16.6, the above tree equals

$$\mathit{splitLeaf}\left(\mathit{huffman}\quad\boxed{\begin{array}{c}c\\w_c\end{array}}\quad\boxed{\begin{array}{c}a\\w_a{+}w_b\end{array}}\quad\boxed{\begin{array}{c}d\\w_d\end{array}}\quad\cdots\quad\boxed{\begin{array}{c}z\\w_z\end{array}}\right)\ w_a\ a\ w_b\ b$$

To prove that this tree is optimum, it suffices by Lemma 16.5 to show that

$$\mathit{huffman}\quad\boxed{\begin{array}{c}c\\w_c\end{array}}\quad\boxed{\begin{array}{c}a\\w_a{+}w_b\end{array}}\quad\boxed{\begin{array}{c}d\\w_d\end{array}}\quad\cdots\quad\boxed{\begin{array}{c}z\\w_z\end{array}}$$

is optimum, which follows from the induction hypothesis. □

In summary, we have established that the *huffman* program, which constitutes a functional implementation of Huffman's algorithm, constructs a binary tree that represents an optimal binary code for the specified alphabet and frequencies.

Part IV

Priority Queues

17
Priority Queues

ADT *Priority_Queue* =

interface
empty :: *'q*
insert :: *'a* ⇒ *'q* ⇒ *'q*
del_min :: *'q* ⇒ *'q*
get_min :: *'q* ⇒ *'a*

abstraction *mset* :: *'q* ⇒ *'a multiset*
invariant *invar* :: *'q* ⇒ *bool*

specification

invar empty	(*empty-inv*)
mset empty = {#}	(*empty*)
invar q ⟶ *invar* (*insert x q*)	(*insert-inv*)
invar q ⟶ *mset* (*insert x q*) = *mset q* + {#*x*#}	(*insert*)
invar q ∧ *mset q* ≠ {#} ⟶ *invar* (*del_min q*)	(*del_min-inv*)
invar q ∧ *mset q* ≠ {#} ⟶ *mset* (*del_min q*) = *mset q* − {#*get_min q*#}	(*del_min*)
invar q ∧ *mset q* ≠ {#} ⟶ *get_min q* = *Min_mset* (*mset q*)	(*get_min*)

Fig. 17.1. ADT *Priority_Queue*

Mergeable priority queues (see Figure 17.2) provide an additional function *merge* (sometimes: *meld* or *union*) with the obvious functionality.

ADT *Priority_Queue_Merge* = *Priority_Queue* +

interface
$merge :: 'q \Rightarrow 'q \Rightarrow 'q$

specification

$invar\ q_1 \wedge invar\ q_2 \longrightarrow invar\ (merge\ q_1\ q_2)$ (*merge-inv*)

$invar\ q_1 \wedge invar\ q_2 \longrightarrow mset\ (merge\ q_1\ q_2) = mset\ q_1 + mset\ q_2$ (*merge*)

Fig. 17.2. ADT *Priority_Queue_Merge*

17.1 Heaps

A popular implementation technique for priority queues are **heaps**, i.e. trees where the minimal element in each subtree is at the root:

$heap :: 'a\ tree \Rightarrow bool$

$heap\ \langle\rangle = True$
$heap\ \langle l,\ m,\ r\rangle$
$= (heap\ l \wedge heap\ r \wedge (\forall x{\in}set\_tree\ l \cup set\_tree\ r.\ m \leqslant x))$

Function *mset_tree* extracts the multiset of elements from a tree:

$mset\_tree :: 'a\ tree \Rightarrow 'a\ multiset$

$mset\_tree\ \langle\rangle = \{\#\}$
$mset\_tree\ \langle l,\ a,\ r\rangle = \{\#a\#\} + mset\_tree\ l + mset\_tree\ r$

If a heap-based implementation provides a *merge* function (e.g. skew heaps in Chapter 24), then the other functions can in principle be defined like this:

$empty = \langle\rangle$

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

$get\_min\ \langle l,\ a,\ r\rangle = a$

$del\_min\ \langle l,\ a,\ r\rangle = merge\ l\ r$

The definitions of *empty* and *get_min* work for any heap.

Note that the following tempting definition of *merge* is functionally correct but leads to very unbalanced heaps:

$merge\ \langle\rangle\ t = t$
$merge\ t\ \langle\rangle = t$
$merge\ (\langle l_1,\ a_1,\ r_1\rangle =: t_1)\ (\langle l_2,\ a_2,\ r_2\rangle =: t_2)$
$= (\mathit{if}\ a_1 \leqslant a_2\ \mathit{then}\ \langle l_1,\ a_1,\ merge\ r_1\ t_2\rangle\ \mathit{else}\ \langle l_2,\ a_2,\ merge\ t_1\ r_2\rangle)$

Exercise 17.1. Show functional correctness of the above definition of *merge* and prove functional correctness of the implementations of *insert* and *del_min*.

TODO exhibit bad sequence of ops and show that it is bad.

18
Leftist Heaps [1]

Leftist heaps are, as the name suggests, an example of heaps in the sense of Section 17.1 and implement mergeable priority queues. We represent leftist heaps as augmented trees (see Section 4.4) to store the so-called **rank**, the height of the rightmost spine:

type_synonym $'a\ lheap = ('a * nat)\ tree$

$rank :: ('a \times nat)\ tree \Rightarrow nat$
$rank\ \langle\rangle = 0$
$rank\ \langle\_,\ \_,\ r\rangle = rank\ r + 1$

$rk :: ('a \times nat)\ tree \Rightarrow nat$
$rk\ \langle\rangle = 0$
$rk\ \langle\_,\ (\_,\ n),\ \_\rangle = n$

Invariants:

$heap :: ('a \times 'b)\ tree \Rightarrow bool$
$heap\ \langle\rangle = True$
$heap\ \langle l,\ (m,\ \_),\ r\rangle$
$= (heap\ l \wedge heap\ r \wedge (\forall x \in set\_tree\ l \cup set\_tree\ r.\ m \leqslant x))$

$ltree :: ('a \times nat)\ tree \Rightarrow bool$
$ltree\ \langle\rangle = True$

[1] isabelle/src/HOL/Data_Structures/Leftist_Heap.thy

```
ltree ⟨l, (_, n), r⟩
= (n = rank r + 1 ∧ rank r ⩽ rank l ∧ ltree l ∧ ltree r)
```

$$\mathit{ltree}\ t \longrightarrow 2^{\mathit{rank}\ t} \leqslant |t|_1$$

18.1 Implementation of ADT *Priority_Queue_Merge*

The key operation is *merge*:

```
merge :: ('a × nat) tree ⇒ ('a × nat) tree ⇒ ('a × nat) tree
merge ⟨⟩ t = t
merge t ⟨⟩ = t
merge (⟨l₁, (a₁, n₁), r₁⟩ =: t₁) (⟨l₂, (a₂, n₂), r₂⟩ =: t₂)
= (if a₁ ⩽ a₂ then node l₁ a₁ (merge r₁ t₂)
   else node l₂ a₂ (merge t₁ r₂))

node :: ('a × nat) tree ⇒ 'a ⇒ ('a × nat) tree ⇒ ('a × nat) tree
node l a r
= (let rl = rk l; rr = rk r
   in if rr ⩽ rl then ⟨l, (a, rr + 1), r⟩ else ⟨r, (a, rl + 1), l⟩)
```

Why does *merge* terminate? Sum vs lex ord

As shown in Section 17.1, once we have *merge*, the other operations are easily definable. We repeat their definition simply because this chapter employs augmented rather than ordinary trees:

```
empty :: ('a × nat) tree
empty = ⟨⟩

get_min :: ('a × nat) tree ⇒ 'a
get_min ⟨_, (a, _), _⟩ = a

insert :: 'a ⇒ ('a × nat) tree ⇒ ('a × nat) tree
insert x t = merge ⟨⟨⟩, (x, 1), ⟨⟩⟩ t

del_min :: ('a × nat) tree ⇒ ('a × nat) tree
```

$del\_min\ \langle\rangle = \langle\rangle$
$del\_min\ \langle l,\ \_,\ r\rangle = merge\ l\ r$

18.2 Correctness

Abstraction function:

$mset\_tree :: ('a \times 'b)\ tree \Rightarrow 'a\ multiset$

$mset\_tree\ \langle\rangle = \{\#\}$
$mset\_tree\ \langle l,\ (a,\ \_),\ r\rangle = \{\#a\#\} + mset\_tree\ l + mset\_tree\ r$

$set\_tree\ t = set\_mset\ (mset\_tree\ t)$
$mset\_tree\ (merge\ h_1\ h_2) = mset\_tree\ h_1 + mset\_tree\ h_2$
$heap\ h \wedge h \neq \langle\rangle \longrightarrow get\_min\ h = Min\ (set\_tree\ h)$

$ltree\ l \wedge ltree\ r \longrightarrow ltree\ (merge\ l\ r)$
$heap\ l \wedge heap\ r \longrightarrow heap\ (merge\ l\ r)$

TODO t to h?? $invar\ t = (heap\ t \wedge ltree\ t)$

18.3 Running Time Analysis

$t\_merge :: ('a \times nat)\ tree \Rightarrow ('a \times nat)\ tree \Rightarrow nat$

$t\_merge\ \langle\rangle\ t = 1$
$t\_merge\ t\ \langle\rangle = 1$
$t\_merge\ (\langle l_1,\ (a_1,\ n_1),\ r_1\rangle =: t_1)\ (\langle l_2,\ (a_2,\ n_2),\ r_2\rangle =: t_2)$
$= (if\ a_1 \leqslant a_2\ then\ 1 + t\_merge\ r_1\ t_2\ else\ 1 + t\_merge\ t_1\ r_2)$

$t\_merge\ l\ r \leqslant rank\ l + rank\ r + 1$
$ltree\ l \wedge ltree\ r \longrightarrow t\_merge\ l\ r \leqslant \lg |l|_1 + \lg |r|_1 + 1$
$ltree\ t \longrightarrow t\_insert\ x\ t \leqslant \lg |t|_1 + 2$
$ltree\ t \longrightarrow t\_del\_min\ t \leqslant 2 \cdot \lg |t|_1 + 1$

19

Priority Queues via Braun Trees [1]

Recall Braun tress from Chapter 12.

TODO: introduce value (and maybe left and right) for type tree

19.1 Implementation of ADT *Priority_Queue*

$insert :: {'a} \Rightarrow {'a}\ tree \Rightarrow {'a}\ tree$

$insert\ a\ \langle\rangle = \langle\langle\rangle,\ a,\ \langle\rangle\rangle$
$insert\ a\ \langle l,\ x,\ r\rangle$
$= (if\ a < x\ then\ \langle insert\ x\ r,\ a,\ l\rangle\ else\ \langle insert\ a\ r,\ x,\ l\rangle)$

$del\_min :: {'a}\ tree \Rightarrow {'a}\ tree$

$del\_min\ \langle\rangle = \langle\rangle$
$del\_min\ \langle\langle\rangle,\ x,\ r\rangle = \langle\rangle$
$del\_min\ \langle l,\ x,\ r\rangle = (let\ (y,\ l') = del\_left\ l\ in\ sift\_down\ r\ y\ l')$

$sift\_down :: {'a}\ tree \Rightarrow {'a} \Rightarrow {'a}\ tree \Rightarrow {'a}\ tree$

$sift\_down\ \langle\rangle\ a\ \_ = \langle\langle\rangle,\ a,\ \langle\rangle\rangle$
$sift\_down\ \langle\langle\rangle,\ x,\ \_\rangle\ a\ \langle\rangle$
$= (if\ a \leqslant x\ then\ \langle\langle\langle\rangle,\ x,\ \langle\rangle\rangle,\ a,\ \langle\rangle\rangle\ else\ \langle\langle\langle\rangle,\ a,\ \langle\rangle\rangle,\ x,\ \langle\rangle\rangle)$
$sift\_down\ (\langle l_1,\ x_1,\ r_1\rangle =: t_1)\ a\ (\langle l_2,\ x_2,\ r_2\rangle =: t_2)$
$= (if\ a \leqslant x_1 \wedge a \leqslant x_2\ then\ \langle t_1,\ a,\ t_2\rangle$
$\quad else\ if\ x_1 \leqslant x_2\ then\ \langle sift\_down\ l_1\ a\ r_1,\ x_1,\ t_2\rangle$

[1] AFP/Priority_Queue_Braun/Priority_Queue_Braun.thy

```
      else ⟨t₁, x₂, sift_down l₂ a r₂⟩)

del_left :: 'a tree ⇒ 'a × 'a tree
del_left ⟨⟨⟩, x, r⟩ = (x, r)
del_left ⟨l, x, r⟩ = (let (y, l') = del_left l in (y, ⟨r, x, l'⟩))
```

19.2 Correctness

Our Braun trees are heaps in the sense of Section 17.1 and functions *heap* and *mset_tree* are defined on them.

$|insert\ x\ t| = |t| + 1$
$mset\_tree\ (insert\ x\ t) = \{\#x\#\} + mset\_tree\ t$
$set\_tree\ (insert\ x\ t) = \{x\} \cup set\_tree\ t$
$braun\ t \longrightarrow braun\ (insert\ x\ t)$
$heap\ t \longrightarrow heap\ (insert\ x\ t)$

$del\_left\ t = (x,\ t') \wedge t \neq \langle\rangle \longrightarrow mset\_tree\ t = \{\#x\#\} + mset\_tree\ t'$
$del\_left\ t = (x,\ t') \wedge t \neq \langle\rangle \longrightarrow set\_tree\ t = \{x\} \cup set\_tree\ t'$
$del\_left\ t = (x,\ t') \wedge t \neq \langle\rangle \wedge heap\ t \longrightarrow heap\ t'$
$del\_left\ t = (x,\ t') \wedge t \neq \langle\rangle \longrightarrow |t| = |t'| + 1$
$del\_left\ t = (x,\ t') \wedge braun\ t \wedge t \neq \langle\rangle \longrightarrow braun\ t'$

$braun\ \langle l,\ a,\ r\rangle \longrightarrow |sift\_down\ l\ a\ r| = |l| + |r| + 1$
$braun\ \langle l,\ a,\ r\rangle \longrightarrow braun\ (sift\_down\ l\ a\ r)$
$braun\ \langle l,\ a,\ r\rangle \longrightarrow$
$mset\_tree\ (sift\_down\ l\ a\ r) = \{\#a\#\} + (mset\_tree\ l + mset\_tree\ r)$
$braun\ \langle l,\ a,\ r\rangle \longrightarrow$
$set\_tree\ (sift\_down\ l\ a\ r) = \{a\} \cup (set\_tree\ l \cup set\_tree\ r)$
$braun\ \langle l,\ a,\ r\rangle \wedge heap\ l \wedge heap\ r \longrightarrow heap\ (sift\_down\ l\ a\ r)$

$braun\ t \longrightarrow braun\ (del\_min\ t)$
$heap\ t \wedge braun\ t \longrightarrow heap\ (del\_min\ t)$
$braun\ t \longrightarrow |del\_min\ t| = |t| - 1$
$braun\ t \wedge t \neq \langle\rangle \longrightarrow mset\_tree\ (del\_min\ t) = mset\_tree\ t - \{\#value\ t\#\}$

TODO comment on having both set and multiset versions of lemmas
TODO running time

20

Binomial Heaps [1]

TODO: variable renaming: x -> a?

datatype $'a\ tree = Node\ nat\ 'a\ ('a\ tree\ list)$

TODO: reorder arguments: rank last or after value

Syntactic sugar:

$\langle r,\ x,\ ts\rangle \equiv Node\ r\ x\ ts$

$rank\ \langle r,\ x,\ ts\rangle = r \qquad root\ \langle r,\ x,\ ts\rangle = x$

TODO: root -> value (as in Tree.thy)??

type_synonym $'a\ heap = 'a\ tree\ list$

TODO: mv the following defs to Correctness? Nicer syntax for image-mset?

$mset\_tree :: 'a\ tree \Rightarrow 'a\ multiset$

$mset\_tree\ \langle\_,\ a,\ ts\rangle$
$= \{\#a\#\} + \bigcup\#\ (image\_mset\ mset\_tree\ (mset\ ts))$

[1] isabelle/src/HOL/Data_Structures/Binomial_Heap.thy

$mset\_heap :: 'a\ tree\ list \Rightarrow 'a\ multiset$

$mset\_heap\ ts = \bigcup\#\ (image\_mset\ mset\_tree\ (mset\ ts))$

Invariants:

$invar\_btree :: 'a\ tree \Rightarrow bool$

$invar\_btree\ \langle r,\ \_,\ ts\rangle$
$= ((\forall\, t{\in}set\ ts.\ invar\_btree\ t) \wedge map\ rank\ ts = rev\ [0..{<}r])$

$invar\_bheap :: 'a\ tree\ list \Rightarrow bool$

$invar\_bheap\ ts$
$= ((\forall\, t{\in}set\ ts.\ invar\_btree\ t) \wedge sorted\_wrt\ ({<})\ (map\ rank\ ts))$

$invar\_otree :: 'a\ tree \Rightarrow bool$

$invar\_otree\ \langle\_,\ x,\ ts\rangle = (\forall\, t{\in}set\ ts.\ invar\_otree\ t \wedge x \leqslant root\ t)$

$invar\_oheap :: 'a\ tree\ list \Rightarrow bool$

$invar\_oheap\ ts = (\forall\, t{\in}set\ ts.\ invar\_otree\ t)$

$invar :: 'a\ tree\ list \Rightarrow bool$

$invar\ ts = (invar\_bheap\ ts \wedge invar\_oheap\ ts)$

TODO shorter names for invar-X, eg (is-)X? TODO: sorted-wrt defined?

20.1 Implementation of ADT *Priority_Queue_Merge*

$link :: 'a\ tree \Rightarrow 'a\ tree \Rightarrow 'a\ tree$

$link\ (\langle r,\ x_1,\ ts_1\rangle =: t_1)\ (\langle r',\ x_2,\ ts_2\rangle =: t_2)$
$= (if\ x_1 \leqslant x_2\ then\ \langle r + 1,\ x_1,\ t_2\ \#\ ts_1\rangle\ else\ \langle r + 1,\ x_2,\ t_1\ \#\ ts_2\rangle)$

TODO: define empty; use min on lists?

$empty = []$

```
get_min :: 'a tree list ⇒ 'a
get_min [t] = root t
get_min (t # ts) = min (root t) (get_min ts)

insert :: 'a ⇒ 'a tree list ⇒ 'a tree list
insert x ts = ins_tree ⟨0, x, []⟩ ts

ins_tree :: 'a tree ⇒ 'a tree list ⇒ 'a tree list
ins_tree t [] = [t]
ins_tree t₁ (t₂ # ts)
= (if rank t₁ < rank t₂ then t₁ # t₂ # ts else ins_tree (link t₁ t₂) ts)
```

```
del_min :: 'a tree list ⇒ 'a tree list
del_min ts
= (case get_min_rest ts of (⟨r, x, ts₁⟩, ts₂) ⇒ merge (rev ts₁) ts₂)

get_min_rest :: 'a tree list ⇒ 'a tree × 'a tree list
get_min_rest [t] = (t, [])
get_min_rest (t # ts)
= (let (t', ts') = get_min_rest ts
    in if root t ⩽ root t' then (t, ts) else (t', t # ts'))

merge :: 'a tree list ⇒ 'a tree list ⇒ 'a tree list
merge ts₁ [] = ts₁
merge [] ts₂ = ts₂
merge (t₁ # ts₁ =: h₁) (t₂ # ts₂ =: h₂)
= (if rank t₁ < rank t₂ then t₁ # merge ts₁ h₂
    else if rank t₂ < rank t₁ then t₂ # merge h₁ ts₂
         else ins_tree (link t₁ t₂) (merge ts₁ ts₂))
```

Why does *merge* terminate? Similar to merge for leftist heap: sum vs lex

20.2 Correctness

$invar\_btree\ t \wedge invar\_bheap\ ts \wedge (\forall t' {\in} set\ ts.\ rank\ t \leqslant rank\ t') \longrightarrow$

$invar\_bheap\ (ins\_tree\ t\ ts)$
$invar\_otree\ t \wedge invar\_oheap\ ts \longrightarrow invar\_oheap\ (ins\_tree\ t\ ts)$
$mset\_heap\ (ins\_tree\ t\ ts) = mset\_tree\ t + mset\_heap\ ts$
$t' \in set\ (ins\_tree\ t\ ts) \wedge (\forall t' {\in} set\ ts.\ rank\ t_0 < rank\ t') \wedge$
$rank\ t_0 < rank\ t \longrightarrow$
$rank\ t_0 < rank\ t'$

$t' \in set\ (merge\ ts_1\ ts_2) \wedge (\forall t_1 {\in} set\ ts_1.\ rank\ t < rank\ t_1) \wedge$
$(\forall t_2 {\in} set\ ts_2.\ rank\ t < rank\ t_2) \longrightarrow$
$rank\ t < rank\ t'$
$invar\_bheap\ ts_1 \wedge invar\_bheap\ ts_2 \longrightarrow invar\_bheap\ (merge\ ts_1\ ts_2)$
$invar\_oheap\ ts_1 \wedge invar\_oheap\ ts_2 \longrightarrow invar\_oheap\ (merge\ ts_1\ ts_2)$
$mset\_heap\ (merge\ ts_1\ ts_2) = mset\_heap\ ts_1 + mset\_heap\ ts_2$

$invar\_otree\ t \wedge x \in\#\ mset\_tree\ t \longrightarrow root\ t \leqslant x$
$ts \neq [] \wedge invar\_oheap\ ts \wedge x \in\#\ mset\_heap\ ts \longrightarrow get\_min\ ts \leqslant x$
$ts \neq [] \longrightarrow get\_min\ ts \in\#\ mset\_heap\ ts$

$ts \neq [] \wedge get\_min\_rest\ ts = (t', ts') \longrightarrow root\ t' = get\_min\ ts$
$get\_min\_rest\ ts = (t', ts') \wedge ts \neq [] \longrightarrow$
$mset\ ts = \{\#t'\#\} + mset\ ts'$
$get\_min\_rest\ ts = (t', ts') \wedge ts \neq [] \wedge invar\_bheap\ ts \longrightarrow$
$invar\_btree\ t'$
$get\_min\_rest\ ts = (t', ts') \wedge ts \neq [] \wedge invar\_bheap\ ts \longrightarrow$
$invar\_bheap\ ts'$
$get\_min\_rest\ ts = (t', ts') \wedge ts \neq [] \wedge invar\_oheap\ ts \longrightarrow$
$invar\_otree\ t'$
$get\_min\_rest\ ts = (t', ts') \wedge ts \neq [] \wedge invar\_oheap\ ts \longrightarrow$
$invar\_oheap\ ts'$

20.3 Running Time Analysis

$invar\_btree\ t \longrightarrow |mset\_tree\ t| = 2^{rank\ t}$
$invar\_bheap\ ts \longrightarrow 2^{|ts|} \leqslant |mset\_heap\ ts| + 1$

TODO: +1 often missing:

```
t_link :: 'a tree ⇒ 'a tree ⇒ nat
t_link _ _ = 1

t_ins_tree :: 'a tree ⇒ 'a tree list ⇒ nat
```

```
$t\_ins\_tree\ \_\ [] = 1$
$t\_ins\_tree\ t_1\ (t_2\ \#\ rest)$
$= (if\ rank\ t_1 < rank\ t_2\ then\ 1$
    $else\ t\_link\ t_1\ t_2 + t\_ins\_tree\ (link\ t_1\ t_2)\ rest)$

$t\_insert :: 'a \Rightarrow 'a\ tree\ list \Rightarrow nat$
$t\_insert\ x\ ts = t\_ins\_tree\ \langle 0, x, []\rangle\ ts$
```

$t\_ins\_tree\ t\ ts \leqslant |ts| + 1$
$invar\ ts \longrightarrow t\_insert\ x\ ts \leqslant \lg\ (|mset\_heap\ ts| + 1) + 1$
$t\_ins\_tree\ t\ ts + |ins\_tree\ t\ ts| = 2 + |ts|$

Thm $invar\ ts \longrightarrow t\_insert\ x\ ts \leqslant \lg\ (|mset\_heap\ ts| + 1) + 1$

```
$t\_merge :: 'a\ tree\ list \Rightarrow 'a\ tree\ list \Rightarrow nat$
$t\_merge\ ts_1\ [] = 1$
$t\_merge\ (t_1\ \#\ ts_1 =: h_1)\ (t_2\ \#\ ts_2 =: h_2)$
$= 1 +$
   $(if\ rank\ t_1 < rank\ t_2\ then\ t\_merge\ ts_1\ h_2$
    $else\ if\ rank\ t_2 < rank\ t_1\ then\ t\_merge\ h_1\ ts_2$
         $else\ t\_ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2) +$
              $t\_merge\ ts_1\ ts_2)$
```

$|merge\ ts_1\ ts_2| + t\_merge\ ts_1\ ts_2 \leqslant 2 \cdot (|ts_1| + |ts_2|) + 1$

Thm $invar\_bheap\ ts_1 \wedge invar\_bheap\ ts_2 \longrightarrow t\_merge\ ts_1\ ts_2 \leqslant 4 \cdot \lg\ (|mset\_heap\ ts_1| + |mset\_heap\ ts_2| + 1) + 2$

```
$t\_get\_min :: 'a\ tree\ list \Rightarrow nat$
$t\_get\_min\ [\_] = 1$
$t\_get\_min\ (\_\ \#\ v\ \#\ va) = 1 + t\_get\_min\ (v\ \#\ va)$

$t\_get\_min\_rest :: 'a\ tree\ list \Rightarrow nat$
$t\_get\_min\_rest\ [\_] = 1$
$t\_get\_min\_rest\ (\_\ \#\ v\ \#\ va) = 1 + t\_get\_min\_rest\ (v\ \#\ va)$

$t\_rev :: 'a\ list \Rightarrow nat$
```

$t\_rev\ xs = |xs| + 1$

$t\_del\_min :: {'a}\ tree\ list \Rightarrow nat$

$t\_del\_min\ ts$
$= t\_get\_min\_rest\ ts\ +$
$\quad (case\ get\_min\_rest\ ts\ of$
$\quad\ (\langle xa,\ x,\ ts_1\rangle,\ ts_2) \Rightarrow t\_rev\ ts_1 + t\_merge\ (rev\ ts_1)\ ts_2)$

$ts \neq [] \longrightarrow t\_get\_min\ ts = |ts|$
$invar\ ts \wedge ts \neq [] \longrightarrow t\_get\_min\ ts \leqslant \lg\ (|mset\_heap\ ts| + 1)$

$ts \neq [] \longrightarrow t\_get\_min\_rest\ ts = |ts|$
$invar\_bheap\ ts \wedge ts \neq [] \longrightarrow t\_get\_min\_rest\ ts \leqslant \lg\ (|mset\_heap\ ts| + 1)$

$invar\_bheap\ (rev\ ts) \longrightarrow t\_rev\ ts \leqslant 1 + \lg\ (|mset\_heap\ ts| + 1)$
$invar\_bheap\ ts \wedge ts \neq [] \longrightarrow t\_del\_min\ ts \leqslant 6 \cdot \lg\ (|mset\_heap\ ts| + 1) + 3$

Part V

Advanced Design and Analysis Techniques

21

Dynamic Programming (Simon Wimmer)

22

Amortized Analysis

[22]

22.1 Introductory Examples

22.2 Real Time Queue

23

Splay Trees

23.1 Implementation [1]

```
insert :: 'a ⇒ 'a tree ⇒ 'a tree
insert x t
= (if t = ⟨⟩ then ⟨⟨⟩, x, ⟨⟩⟩
   else case splay x t of
          ⟨l, a, r⟩ ⇒ case cmp x a of
                         LT ⇒ ⟨l, x, ⟨⟨⟩, a, r⟩⟩ |
                         EQ ⇒ ⟨l, a, r⟩ |
                         GT ⇒ ⟨⟨l, a, ⟨⟩⟩, x, r⟩)
```

Fix problem with "xa" in splaymax last line:

23.2 Correctness

23.3 Amortized Analysis [2]

```
t_splay :: 'a ⇒ 'a tree ⇒ nat
t_splay _ ⟨⟩ = 1
t_splay x ⟨AB, b, CD⟩
```

[1] AFP/Splay_Tree/Splay_Tree.thy
[2] AFP/Amortized_Complexity/Splay_Tree_Analysis.thy

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

Fig. 23.1. Function *splay*

```
$=$ (case $cmp\ x\ b$ of
    $LT \Rightarrow$ case $AB$ of
            $\langle\rangle \Rightarrow 1$ |
            $\langle A,\ a,\ B\rangle \Rightarrow$ case $cmp\ x\ a$ of
                            $LT \Rightarrow$ if $A = \langle\rangle$ then $1$ else $t\_splay\ x\ A + 1$ |
                            $EQ \Rightarrow 1$ |
                            $GT \Rightarrow$ if $B = \langle\rangle$ then $1$ else $t\_splay\ x\ B + 1$ |
    $EQ \Rightarrow 1$ |
    $GT \Rightarrow$ case $CD$ of
            $\langle\rangle \Rightarrow 1$ |
            $\langle C,\ c,\ D\rangle \Rightarrow$ case $cmp\ x\ c$ of
                            $LT \Rightarrow$ if $C = \langle\rangle$ then $1$ else $t\_splay\ x\ C + 1$ |
                            $EQ \Rightarrow 1$ |
splay x ⟨⟩ = ⟨⟩
splay x ⟨A, x, B⟩ = ⟨A, x, B⟩
x < b ⟶ splay x ⟨⟨A, x, B⟩, b, C⟩ = ⟨A, x, ⟨B, b, C⟩⟩
x < b ⟶ splay x ⟨⟨⟩, b, A⟩ = ⟨⟨⟩, b, A⟩
x < a ∧ x < b ⟶ splay x ⟨⟨⟨⟩, a, A⟩, b, B⟩ = ⟨⟨⟩, a, ⟨A, b, B⟩⟩
x < b ∧ x < c ∧ AB ≠ ⟨⟩ ⟶
splay x ⟨⟨AB, b, C⟩, c, D⟩ = (case splay x AB of
                                ⟨A, a, B⟩ ⇒ ⟨A, a, ⟨B, b, ⟨C, c, D⟩⟩⟩)
a < x ∧ x < b ⟶ splay x ⟨⟨A, a, ⟨⟩⟩, b, B⟩ = ⟨A, a, ⟨⟨⟩, b, B⟩⟩
a < x ∧ x < c ∧ BC ≠ ⟨⟩ ⟶
splay x ⟨⟨A, a, BC⟩, c, D⟩ = (case splay x BC of
                                ⟨B, b, C⟩ ⇒ ⟨⟨A, a, B⟩, b, ⟨C, c, D⟩⟩)
a < x ⟶ splay x ⟨A, a, ⟨B, x, C⟩⟩ = ⟨⟨A, a, B⟩, x, C⟩
a < x ⟶ splay x ⟨A, a, ⟨⟩⟩ = ⟨A, a, ⟨⟩⟩
a < x ∧ x < c ∧ BC ≠ ⟨⟩ ⟶
splay x ⟨A, a, ⟨BC, c, D⟩⟩ = (case splay x BC of
                                ⟨B, b, C⟩ ⇒ ⟨⟨A, a, B⟩, b, ⟨C, c, D⟩⟩)
a < x ∧ x < b ⟶ splay x ⟨A, a, ⟨⟨⟩, b, C⟩⟩ = ⟨⟨A, a, ⟨⟩⟩, b, C⟩
a < x ∧ b < x ⟶ splay x ⟨A, a, ⟨B, b, ⟨⟩⟩⟩ = ⟨⟨A, a, B⟩, b, ⟨⟩⟩
a < x ∧ b < x ∧ CD ≠ ⟨⟩ ⟶
splay x ⟨A, a, ⟨B, b, CD⟩⟩ = (case splay x CD of
                                ⟨C, x, xa⟩ ⇒ ⟨⟨⟨A, a, B⟩, b, C⟩, x, xa⟩)
```

Fig. 23.2. Conditional definition of function *splay*

```
delete :: 'a ⇒ 'a tree ⇒ 'a tree

delete x t
= (if t = ⟨⟩ then ⟨⟩
   else case splay x t of
        ⟨l, a, r⟩ ⇒
          if x ≠ a then ⟨l, a, r⟩
          else if l = ⟨⟩ then r
               else case splay_max l of ⟨l', m, r'⟩ ⇒ ⟨l', m, r⟩)

splay_max :: 'a tree ⇒ 'a tree

splay_max ⟨⟩ = ⟨⟩
splay_max ⟨A, a, ⟨⟩⟩ = ⟨A, a, ⟨⟩⟩
splay_max ⟨A, a, ⟨B, b, CD⟩⟩
= (if CD = ⟨⟩ then ⟨⟨A, a, B⟩, b, ⟨⟩⟩
   else case splay_max CD of ⟨C, x, xa⟩ ⇒ ⟨⟨⟨A, a, B⟩, b, C⟩, x, xa⟩)
```

Fig. 23.3. Functions *insert*, *delete* and *splay_max*

$$GT \Rightarrow \textit{if } D = \langle\rangle \textit{ then } 1 \textit{ else } t\_splay\ x\ D + 1)$$

$t \neq \langle\rangle \wedge bst\ t \longrightarrow$
$(\exists\, a' \in set\_tree\ t.$
$\quad splay\ a'\ t = splay\ a\ t \wedge t\_splay\ a'\ t = t\_splay\ a\ t)$

$\Phi :: {'a}\ tree \Rightarrow real$

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

$\varphi\ t = \lg |t|_1$

$a\_splay\ a\ t = t\_splay\ a\ t + \Phi\ (splay\ a\ t) - \Phi\ t$

$bst\ t \wedge \langle l,\ x,\ r\rangle \in subtrees\ t \longrightarrow$
$a\_splay\ x\ t \leqslant 3 \cdot (\varphi\ t - \varphi\ \langle l,\ x,\ r\rangle) + 1$
$bst\ t \wedge a \in set\_tree\ t \longrightarrow a\_splay\ a\ t \leqslant 3 \cdot (\varphi\ t - 1) + 1$
$bst\ t \longrightarrow a\_splay\ a\ t \leqslant 3 \cdot \varphi\ t + 1$

$bst\ t \longrightarrow t\_splay\ a\ t + \Phi\ (insert\ a\ t) - \Phi\ t \leqslant 4 \cdot \varphi\ t + 2$

$a\_splay\_max\ t = t\_splay\_max\ t + \Phi\ (splay\_max\ t) - \Phi\ t$

$bst\ t \wedge t \neq \langle\rangle \longrightarrow a\_splay\_max\ t \leqslant 3 \cdot (\varphi\ t - 1) + 1$
$bst\ t \longrightarrow a\_splay\_max\ t \leqslant 3 \cdot \varphi\ t + 1$

$bst\ t \longrightarrow t\_delete\ a\ t + \Phi\ (delete\ a\ t) - \Phi\ t \leqslant 6 \cdot \varphi\ t + 2$

24

Skew Heaps

TODO: unify variable names: h or t

24.1 Implementation of ADT *Priority_Queue_Merge* [1]

TODO Compare and unify with Leftist heap defs and proofs

Skew heaps are, as the name suggests, an example of heaps in the sense of Section 17.1 and implement mergeable priority queues. The key operation is *merge*:

$$\mathit{merge} :: {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree} \Rightarrow {'a}\ \mathit{tree}$$

$$\mathit{merge}\ \langle\rangle\ h = h$$
$$\mathit{merge}\ h\ \langle\rangle = h$$
$$\mathit{merge}\ (\langle l_1,\ a_1,\ r_1\rangle =: h_1)\ (\langle l_2,\ a_2,\ r_2\rangle =: h_2)$$
$$= (\mathit{if}\ a_1 \leqslant a_2\ \mathit{then}\ \langle \mathit{merge}\ h_2\ r_1,\ a_1,\ l_1\rangle\ \mathit{else}\ \langle \mathit{merge}\ h_1\ r_2,\ a_2,\ l_2\rangle)$$

The remaining operations are defined as in Section 17.1.

24.2 Correctness

$$|\mathit{merge}\ t_1\ t_2| = |t_1| + |t_2|$$
$$\mathit{mset\_tree}\ (\mathit{merge}\ h_1\ h_2) = \mathit{mset\_tree}\ h_1 + \mathit{mset\_tree}\ h_2$$
$$\mathit{set\_tree}\ (\mathit{merge}\ h_1\ h_2) = \mathit{set\_tree}\ h_1 \cup \mathit{set\_tree}\ h_2$$
$$\mathit{heap}\ h_1 \wedge \mathit{heap}\ h_2 \longrightarrow \mathit{heap}\ (\mathit{merge}\ h_1\ h_2)$$

[1] AFP/Skew_Heap/Skew_Heap.thy

24.3 Amortized Analysis [2]

TODO improve layout:

```
t_merge :: 'a tree ⇒ 'a tree ⇒ nat
t_merge ⟨⟩ _ = 1
t_merge _ ⟨⟩ = 1
t_merge ⟨l₁, a₁, r₁⟩ ⟨l₂, a₂, r₂⟩
= (if a₁ ≤ a₂ then t_merge ⟨l₂, a₂, r₂⟩ r₁
   else t_merge ⟨l₁, a₁, r₁⟩ r₂) +
  1
```

```
Φ :: 'a tree ⇒ int
Φ ⟨⟩ = 0
Φ ⟨l, _, r⟩ = Φ l + Φ r + rh l r

rh :: 'a tree ⇒ 'a tree ⇒ nat
rh l r = (if |l| < |r| then 1 else 0)
```

```
lrh :: 'a tree ⇒ nat
lrh ⟨⟩ = 0
lrh ⟨l, _, r⟩ = rh l r + lrh l

rlh :: 'a tree ⇒ nat
rlh ⟨⟩ = 0
rlh ⟨l, _, r⟩ = 1 − rh l r + rlh r
```

$$2^{lrh\ h} \leqslant |h| + 1 \qquad lrh\ h \leqslant \lg |h|_1$$
$$2^{rlh\ h} \leqslant |h| + 1 \qquad rlh\ h \leqslant \lg |h|_1$$

$$t\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$$
$$\leqslant lrh\ (merge\ t_1\ t_2) + rlh\ t_1 + rlh\ t_2 + 1$$
$$t\_merge\ t_1\ t_2 + \Phi\ (merge\ t_1\ t_2) - \Phi\ t_1 - \Phi\ t_2$$
$$\leqslant 3 \cdot \lg\ (|t_1|_1 + |t_2|_1) + 1$$

TODO show proof of amerge

[2] AFP/Amortized_Complexity/Skew_Heap_Analysis.thy

```
t_insert :: 'a ⇒ 'a tree ⇒ int
t_insert a h = t_merge ⟨⟨⟩, a, ⟨⟩⟩ h + 1

t_del_min :: 'a tree ⇒ int
t_del_min h = (case h of ⟨⟩ ⇒ 1 | ⟨t₁, a, t₂⟩ ⇒ t_merge t₁ t₂ + 1)
```

$$t\_insert\ a\ h + \Phi\ (insert\ a\ h) - \Phi\ h \leqslant 3 \cdot \lg\ (|h|_1 + 2) + 2$$
$$t\_del\_min\ h + \Phi\ (del\_min\ h) - \Phi\ h \leqslant 3 \cdot \lg\ (|h|_1 + 2) + 2$$

25

Pairing Heaps

25.1 Implementations via Lists [1]

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

$mset\_heap :: 'a\ heap \Rightarrow 'a\ multiset$

$mset\_heap\ Empty = \{\#\}$
$mset\_heap\ (Hp\ x\ hs) = \{\#x\#\} + \bigcup\#\ (mset\ (map\ mset\_heap\ hs))$

$pheap :: 'a\ heap \Rightarrow bool$

$pheap\ Empty = True$
$pheap\ (Hp\ x\ hs)$
$= (\forall\, h{\in}set\ hs.\ (\forall\, y{\in}\#mset\_heap\ h.\ x \leqslant y) \wedge pheap\ h)$

Note: Empty only at the top.

$merge :: 'a\ heap \Rightarrow 'a\ heap \Rightarrow 'a\ heap$

$merge\ h\ Empty = h$
$merge\ Empty\ (Hp\ v\ va) = Hp\ v\ va$
$merge\ (Hp\ x\ lx =: hx)\ (Hp\ y\ ly =: hy)$
$= (\mathit{if}\ x < y\ \mathit{then}\ Hp\ x\ (hy\ \#\ lx)\ \mathit{else}\ Hp\ y\ (hx\ \#\ ly))$

$empty = Empty$

[1] AFP/Pairing_Heap/Pairing_Heap_List1.thy

```
get_min :: 'a heap ⇒ 'a
get_min (Hp x _) = x

insert :: 'a ⇒ 'a heap ⇒ 'a heap
insert x h = merge (Hp x []) h
```

```
del_min :: 'a heap ⇒ 'a heap
del_min Empty = Empty
del_min (Hp x hs) = merge_pairs hs

merge_pairs :: 'a heap list ⇒ 'a heap
merge_pairs [] = Empty
merge_pairs [h] = h
merge_pairs (h₁ # h₂ # hs)
= merge (merge h₁ h₂) (merge_pairs hs)
```

25.1.1 Correctness

$pheap\ h_1 \wedge pheap\ h_2 \longrightarrow pheap\ (merge\ h_1\ h_2)$
$(\forall\, h{\in}set\ hs.\ pheap\ h) \longrightarrow pheap\ (merge\_pairs\ hs)$
$pheap\ h \longrightarrow pheap\ (del\_min\ h)$

$h \neq Empty \longrightarrow get\_min\ h \in\#\ mset\_heap\ h$
$h \neq Empty \wedge pheap\ h \wedge x \in\#\ mset\_heap\ h \longrightarrow get\_min\ h \leqslant x$
$mset\_heap\ (merge\ h_1\ h_2) = mset\_heap\ h_1 + mset\_heap\ h_2$
$mset\_heap\ (merge\_pairs\ hs)$
$= \bigcup\#\ (image\_mset\ mset\_heap\ (mset\ hs))$
$h \neq Empty \longrightarrow$
$mset\_heap\ (del\_min\ h) = mset\_heap\ h - \{\#get\_min\ h\#\}$

25.2 Implementations via Trees [2]

[2] AFP/Pairing_Heap/Pairing_Heap_Tree.thy

```
empty = ⟨⟩

get_min :: 'a tree ⇒ 'a
get_min ⟨_, x, _⟩ = x

link :: 'a tree ⇒ 'a tree
link ⟨⟩ = ⟨⟩
link ⟨lx, x, ⟨⟩⟩ = ⟨lx, x, ⟨⟩⟩
link ⟨lx, x, ⟨ly, y, ry⟩⟩
= (if x < y then ⟨⟨ly, y, lx⟩, x, ry⟩ else ⟨⟨lx, x, ly⟩, y, ry⟩)

pass₁ :: 'a tree ⇒ 'a tree
pass₁ ⟨⟩ = ⟨⟩
pass₁ ⟨lx, x, ⟨⟩⟩ = ⟨lx, x, ⟨⟩⟩
pass₁ ⟨lx, x, ⟨ly, y, ry⟩⟩ = link ⟨lx, x, ⟨ly, y, pass₁ ry⟩⟩

pass₂ :: 'a tree ⇒ 'a tree
pass₂ ⟨⟩ = ⟨⟩
pass₂ ⟨l, x, r⟩ = link ⟨l, x, pass₂ r⟩

get_min :: 'a tree ⇒ 'a
get_min ⟨_, x, _⟩ = x

merge :: 'a tree ⇒ 'a tree ⇒ 'a tree
merge ⟨⟩ h = h
merge h ⟨⟩ = h
merge ⟨lx, x, ⟨⟩⟩ ⟨ly, y, ⟨⟩⟩ = link ⟨lx, x, ⟨ly, y, ⟨⟩⟩⟩

insert :: 'a ⇒ 'a tree ⇒ 'a tree
insert x h = merge ⟨⟨⟩, x, ⟨⟩⟩ h
```

25.3 Amortized Analysis [3]

$\Phi :: 'a\ tree \Rightarrow real$

$\Phi\ \langle\rangle = 0$

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

$is\_root :: 'a\ tree \Rightarrow bool$

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

$len :: 'a\ tree \Rightarrow nat$

$len\ \langle\rangle = 0$

$len\ \langle\_,\ \_,\ r\rangle = 1 + len\ r$

$|link\ h| = |h|$
$|pass_1\ h| = |h|$
$|pass_2\ h| = |h|$
$is\_root\ h_1 \wedge is\_root\ h_2 \longrightarrow |merge\ h_1\ h_2| = |h_1| + |h_2|$

$is\_root\ h \longrightarrow \Phi\ (insert\ x\ h) - \Phi\ h \leqslant \lg\ (|h| + 1)$
$h_1 = \langle lx,\ x,\ \langle\rangle\rangle \wedge h_2 = \langle ly,\ y,\ \langle\rangle\rangle \longrightarrow$
$\Phi\ (merge\ h_1\ h_2) - \Phi\ h_1 - \Phi\ h_2 \leqslant \lg\ (|h_1| + |h_2|) + 1$

$upperbound :: 'a\ tree \Rightarrow real$

$upperbound\ \langle\rangle = 0$

$upperbound\ \langle\_,\ \_,\ \langle\rangle\rangle = 0$

$upperbound\ \langle lx,\ \_,\ \langle ly,\ \_,\ \langle\rangle\rangle\rangle = 2 \cdot \lg\ (|lx| + |ly| + 2)$

$upperbound\ \langle lx,\ \_,\ \langle ly,\ \_,\ ry\rangle\rangle$
$= 2 \cdot \lg\ (|lx| + |ly| + |ry| + 2) - 2 \cdot \lg\ |ry| - 2 + upperbound\ ry$

$\Phi\ (pass_1\ hs) - \Phi\ hs \leqslant upperbound\ hs$
$hs \neq \langle\rangle \longrightarrow \Phi\ (pass_1\ hs) - \Phi\ hs \leqslant 2 \cdot \lg\ |hs| - len\ hs + 2$
$hs \neq \langle\rangle \longrightarrow \Phi\ (pass_2\ hs) - \Phi\ hs \leqslant \lg\ |hs|$
$hs \neq \langle\rangle \longrightarrow \Phi\ (merge\_pairs\ hs) - \Phi\ hs \leqslant 3 \cdot \lg\ |hs| - len\ hs + 2$
$lx \neq \langle\rangle \longrightarrow$
$\Phi\ (del\_min\ \langle lx,\ x,\ \langle\rangle\rangle) - \Phi\ \langle lx,\ x,\ \langle\rangle\rangle \leqslant 3 \cdot \lg\ |lx| - len\ lx + 2$

TODO explicit amortized statements

[3] AFP/Amortized_Complexity/Pairing_Heap_Tree_Analysis.thy

Part VI

Appendix

A

List Library

The following functions on lists are predefined:

$length :: 'a\ list \Rightarrow nat$

$|[]| = 0$
$|x\ \#\ xs| = |xs| + 1$

$set :: 'a\ list \Rightarrow 'a\ set$

$set\ [] = \{\}$
$set\ (x\ \#\ xs) = \{x\} \cup set\ xs$

$map :: ('a \Rightarrow 'aa) \Rightarrow 'a\ list \Rightarrow 'aa\ list$

$map\ f\ [] = []$
$map\ f\ (x\ \#\ xs) = f\ x\ \#\ map\ f\ xs$

$(@) :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$

$[]\ @\ ys = ys$
$(x\ \#\ xs)\ @\ ys = x\ \#\ xs\ @\ ys$

$filter :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$

$filter\ \_\ [] = []$
$filter\ P\ (x\ \#\ xs) = (if\ P\ x\ then\ x\ \#\ filter\ P\ xs\ else\ filter\ P\ xs)$

$take :: nat \Rightarrow 'a\ list \Rightarrow 'a\ list$

$take\ \_\ [] = []$
$take\ n\ (x\ \#\ xs) = (case\ n\ of\ 0 \Rightarrow []\ |\ m + 1 \Rightarrow x\ \#\ take\ m\ xs)$

$drop :: nat \Rightarrow 'a\ list \Rightarrow 'a\ list$

$drop\ \_\ \ [] = []$
$drop\ n\ (x\ \#\ xs) = (case\ n\ of\ 0 \Rightarrow x\ \#\ xs\ |\ m + 1 \Rightarrow drop\ m\ xs)$

$hd :: 'a\ list \Rightarrow 'a$

$hd\ (x\ \#\ xs) = x$

$tl :: 'a\ list \Rightarrow 'a\ list$

$tl\ [] = []$
$tl\ (x\ \#\ xs) = xs$

$butlast :: 'a\ list \Rightarrow 'a\ list$

$butlast\ [] = []$
$butlast\ (x\ \#\ xs) = (if\ xs = []\ then\ []\ else\ x\ \#\ butlast\ xs)$

$(!) :: 'a\ list \Rightarrow nat \Rightarrow 'a$

$(x\ \#\ xs)\ !\ n = (case\ n\ of\ 0 \Rightarrow x\ |\ k + 1 \Rightarrow xs\ !\ k)$

$list\_update :: 'a\ list \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ list$

$[][\_ := \_] = []$
$(x\ \#\ xs)[i := v] = (case\ i\ of\ 0 \Rightarrow v\ \#\ xs\ |\ j + 1 \Rightarrow x\ \#\ xs[j := v])$

$upt :: nat \Rightarrow nat \Rightarrow nat\ list$

$[\_ ..{<}0] = []$
$[i..{<}j + 1] = (if\ i \leqslant j\ then\ [i..{<}j]\ @\ [j]\ else\ [])$

$replicate :: nat \Rightarrow 'a \Rightarrow 'a\ list$

$replicate\ 0\ \_\ = []$
$replicate\ (n + 1)\ x = x\ \#\ replicate\ n\ x$

$sum\_list :: 'a\ list \Rightarrow 'a$

$sum\_list\ [] = 0$
$sum\_list\ (x\ \#\ xs) = x + sum\_list\ xs$

References

1. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962. Translated from Russian by M.J. Ricci.
2. A. Appel. Efficient verified red-black trees. Unpublished, 2011.
3. C. Ballarin. *Tutorial to Locales and Locale Interpretation*. https://isabelle.in.tum.de/doc/locales.pdf.
4. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.
5. J. C. Blanchette. Proof pearl: Mechanizing the textbook proof of Huffman's algorithm in Isabelle/HOL. *J. Autom. Reasoning*, 43(1):1–18, 2009.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.
7. R. 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. ACM, 1959.
8. M. Eberl. The number of comparisons in quicksort. *Archive of Formal Proofs*, Mar. 2017. http://isa-afp.org/entries/Quick_Sort_Cost.html, Formal proof development.
9. M. Eberl, M. W. Haslbeck, and T. Nipkow. Verified analysis of random binary tree structures. In J. Avigad and A. Mahboubi, editors, *Interactive Theorem Proving (ITP 2018)*, volume 10895 of *LNCS*, pages 196–214. Springer, 2018.
10. J.-C. Filliâtre and P. Letouzey. Functors for proofs and programs. In *ESOP*, volume 2986 of *LNCS*, pages 370–384. Springer, 2004.
11. E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.
12. K. Germane and M. Might. Deletion: The curse of the red-black tree. *J. Functional Programming*, 24(4):423–433, 2014.
13. L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (FOCS 1978)*, pages 8–21, 1978.
14. F. Haftmann. *Haskell-style type classes with Isabelle/Isar*. http://isabelle.in.tum.de/doc/classes.pdf.

15. R. 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.
16. D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E.*, pages 1098–1101, 1952.
17. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 2nd edition, 1990.
18. S. Kahrs. Red black trees with types. *J. Functional Programming*, 11(4):425–432, 2001.
19. D. E. Knuth. *The Art of Computer Programming, vol. 1: Fundamental Algorithms*. Addison–Wesley, 3rd edition, 1997.
20. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
21. D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
22. T. Nipkow. Amortized complexity verified. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pages 310–324. Springer, 2015.
23. T. Nipkow. Automatic functional correctness proofs for functional search trees. In J. Blanchette and S. Merz, editors, *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *LNCS*, pages 307–322. Springer, 2016.
24. T. Nipkow and T. Sewell. Proof pearl: Braun trees. In J. Blanchette and C. Hritcu, editors, *Certified Programs and Proofs, CPP 2020*, pages ?–? ACM, 2020.
25. C. Okasaki. Three algorithms on Braun trees. *J. Functional Programming*, 7(6):661–666, 1997.
26. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
27. M. Rem and W. Braun. A logarithmic implementation of flexible arrays. Memorandum MR83/4. Eindhoven University of Techology, 1983.
28. L. Théry. Formalising Huffman's algorithm. Technical Report TRCS 034, Department of Informatics, University of L'Aquila, 2004.
29. B. Zhan. Efficient verification of imperative programs using auto2. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018*, volume 10805 of *LNCS*, pages 23–40. Springer, 2018.