Functional Data Structures

Tobias Nipkow

June 9, 2019

Abstract

A collection of verified functional data structures. The emphasis is on conciseness of algorithms and succinctness of proofs, more in the style of a textbook than a library of efficient algorithms.

For more details see [12].

Contents

1 Sorting

theory *Sorting*
imports
Complex_Main
HOL−Library.Multiset
begin

hide_const *List.insort*

declare *Let_def* [*simp*]

1.1 Insertion Sort

fun *insort* :: *'a::linorder* ⇒ *'a list* ⇒ *'a list* **where**
insort x [] = [*x*] |
insort x (*y*#*ys*) =
(*if x* ≤ *y then x*#*y*#*ys else y*#(*insort x ys*))

fun *isort* :: *'a::linorder list* ⇒ *'a list* **where**
isort [] = [] |
isort (*x*#*xs*) = *insort x* (*isort xs*)

1.1.1 Functional Correctness

lemma *mset_insort*: *mset* (*insort x xs*) = *add_mset x* (*mset xs*)
apply(*induction xs*)
apply *auto*
done

lemma *mset_isort*: *mset* (*isort xs*) = *mset xs*
apply(*induction xs*)
apply *simp*
apply (*simp add*: *mset_insort*)
done

lemma *set_insort*: *set* (*insort x xs*) = *insert x* (*set xs*)
by (*metis mset_insort set_mset_add_mset_insert set_mset_mset*)

lemma *sorted_insort*: *sorted* (*insort a xs*) = *sorted xs*
apply(*induction xs*)
apply(*auto simp add*: *set_insort*)
done

lemma *sorted_isort*: *sorted* (*isort xs*)

apply(*induction xs*)
apply(*auto simp*: *sorted_insort*)
done

1.1.2 Time Complexity

We count the number of function calls.

insort x [] = [x] insort x (y#ys) = (if x ≤ y then x#y#ys else y#(insort x ys))

fun *t_insort* :: *'a::linorder ⇒ 'a list ⇒ nat* **where**
t_insort x [] = 1 |
t_insort x (y#ys) =
(if x ≤ y then 0 else t_insort x ys) + 1

isort [] = [] isort (x#xs) = insort x (isort xs)

fun *t_isort* :: *'a::linorder list ⇒ nat* **where**
t_isort [] = 1 |
t_isort (x#xs) = t_isort xs + t_insort x (isort xs) + 1

lemma *t_insort_length*: *t_insort x xs ≤ length xs + 1*
apply(*induction xs*)
apply *auto*
done

lemma *length_insort*: *length (insort x xs) = length xs + 1*
apply(*induction xs*)
apply *auto*
done

lemma *length_isort*: *length (isort xs) = length xs*
apply(*induction xs*)
apply (*auto simp*: *length_insort*)
done

lemma *t_isort_length*: *t_isort xs ≤ (length xs + 1) ^ 2*
proof(*induction xs*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons x xs*)
have *t_isort (x#xs) = t_isort xs + t_insort x (isort xs) + 1* **by** *simp*
also have *... ≤ (length xs + 1) ^ 2 + t_insort x (isort xs) + 1*
using *Cons.IH* **by** *simp*

also have ... $\leq$ (*length xs* + *1*) ^ *2* + *length xs* + *1* + *1*
using *t_insort_length*[*of x isort xs*] **by** (*simp add*: *length_isort*)
also have ... $\leq$ (*length*(*x*#*xs*) + *1*) ^ *2*
by (*simp add*: *power2_eq_square*)
finally show *?case* **.**
qed

1.2 Merge Sort

fun *merge* :: *'a*::*linorder list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**
merge [] *ys* = *ys* |
merge xs [] = *xs* |
merge (*x*#*xs*) (*y*#*ys*) = (*if x* $\leq$ *y then x* # *merge xs* (*y*#*ys*) *else y* # *merge* (*x*#*xs*) *ys*)

fun *msort* :: *'a*::*linorder list* $\Rightarrow$ *'a list* **where**
msort xs = (*let n* = *length xs in*
if n $\leq$ *1 then xs*
else merge (*msort* (*take* (*n div 2*) *xs*)) (*msort* (*drop* (*n div 2*) *xs*)))

declare *msort.simps* [*simp del*]

1.2.1 Functional Correctness

lemma *mset_merge*: *mset*(*merge xs ys*) = *mset xs* + *mset ys*
by(*induction xs ys rule*: *merge.induct*) *auto*

lemma *mset_msort*: *mset* (*msort xs*) = *mset xs*
proof(*induction xs rule*: *msort.induct*)
case (*1 xs*)
let *?n* = *length xs*
let *?ys* = *take* (*?n div 2*) *xs*
let *?zs* = *drop* (*?n div 2*) *xs*
show *?case*
proof *cases*
assume *?n* $\leq$ *1*
thus *?thesis* **by**(*simp add*: *msort.simps*[*of xs*])
next
assume ¬ *?n* $\leq$ *1*
hence *mset* (*msort xs*) = *mset* (*msort ?ys*) + *mset* (*msort ?zs*)
by(*simp add*: *msort.simps*[*of xs*] *mset_merge*)
also have ... = *mset ?ys* + *mset ?zs*
using ‹¬ *?n* $\leq$ *1*› **by**(*simp add*: *1.IH*)
also have ... = *mset* (*?ys* @ *?zs*) **by** (*simp del*: *append_take_drop_id*)

```
    also have ... = mset xs by simp
    finally show ?thesis .
  qed
qed
```

Via the previous lemma or directly:

```
lemma set_merge: set(merge xs ys) = set xs ∪ set ys
by (metis mset_merge set_mset_mset set_mset_union)

lemma set(merge xs ys) = set xs ∪ set ys
by(induction xs ys rule: merge.induct) (auto)

lemma sorted_merge: sorted (merge xs ys) ⟷ (sorted xs ∧ sorted ys)
by(induction xs ys rule: merge.induct) (auto simp: set_merge)

lemma sorted_msort: sorted (msort xs)
proof(induction xs rule: msort.induct)
  case (1 xs)
  let ?n = length xs
  show ?case
  proof cases
    assume ?n ≤ 1
    thus ?thesis by(simp add: msort.simps[of xs] sorted01)
  next
    assume ¬ ?n ≤ 1
    thus ?thesis using 1.IH
      by(simp add: sorted_merge msort.simps[of xs])
  qed
qed
```

1.2.2 Time Complexity

We only count the number of comparisons between list elements.

```
fun c_merge :: 'a::linorder list ⇒ 'a list ⇒ nat where
c_merge [] ys = 0 |
c_merge xs [] = 0 |
c_merge (x#xs) (y#ys) = 1 + (if x ≤ y then c_merge xs (y#ys) else
c_merge (x#xs) ys)

lemma c_merge_ub: c_merge xs ys ≤ length xs + length ys
by (induction xs ys rule: c_merge.induct) auto

fun c_msort :: 'a::linorder list ⇒ nat where
c_msort xs =
```

```
  (let n = length xs;
      ys = take (n div 2) xs;
      zs = drop (n div 2) xs
   in if n ≤ 1 then 0
     else c_msort ys + c_msort zs + c_merge (msort ys) (msort zs))

declare c_msort.simps [simp del]

lemma length_merge: length(merge xs ys) = length xs + length ys
apply (induction xs ys rule: merge.induct)
apply auto
done

lemma length_msort: length(msort xs) = length xs
proof (induction xs rule: msort.induct)
  case (1 xs)
  thus ?case by (auto simp: msort.simps[of xs] length_merge)
qed
```

Why structured proof? To have the name "xs" to specialize msort.simps with xs to ensure that msort.simps cannot be used recursively. Also works without this precaution, but that is just luck.

```
lemma c_msort_le: length xs = 2^k ⟹ c_msort xs ≤ k * 2^k
proof(induction k arbitrary: xs)
  case 0 thus ?case by (simp add: c_msort.simps)
next
  case (Suc k)
  let ?n = length xs
  let ?ys = take (?n div 2) xs
  let ?zs = drop (?n div 2) xs
  show ?case
  proof (cases ?n ≤ 1)
    case True
    thus ?thesis by(simp add: c_msort.simps)
  next
    case False
    have c_msort(xs) =
      c_msort ?ys + c_msort ?zs + c_merge (msort ?ys) (msort ?zs)
      by (simp add: c_msort.simps msort.simps)
    also have ... ≤ c_msort ?ys + c_msort ?zs + length ?ys + length ?zs
     using c_merge_ub[of msort ?ys msort ?zs] length_msort[of ?ys] length_msort[of
?zs]
      by arith
    also have ... ≤ k * 2^k + c_msort ?zs + length ?ys + length ?zs
```

```
    using Suc.IH[of ?ys] Suc.prems by simp
  also have ... ≤ k * 2^k + k * 2^k + length ?ys + length ?zs
    using Suc.IH[of ?zs] Suc.prems by simp
  also have ... = 2 * k * 2^k + 2 * 2 ^ k
    using Suc.prems by simp
  finally show ?thesis by simp
 qed
qed
```

```
lemma c_msort_log: length xs = 2^k ⟹ c_msort xs ≤ length xs * log 2
(length xs)
using c_msort_le[of xs k] apply (simp add: log_nat_power algebra_simps)
by (metis (mono_tags) numeral_power_eq_of_nat_cancel_iff of_nat_le_iff of_nat_mult)
```

1.3 Bottom-Up Merge Sort

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

For the termination proof of *merge_all* below.

```
lemma length_merge_adjacent[simp]: length (merge_adj xs) = (length xs +
1) div 2
by (induction xs rule: merge_adj.induct) auto
```

```
fun merge_all :: ('a::linorder) list list ⇒ 'a list where
merge_all [] = [] |
merge_all [xs] = xs |
merge_all xss = merge_all (merge_adj xss)
```

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

1.3.1 Functional Correctness

```
lemma mset_merge_adj:
  ⋃# (image_mset mset (mset (merge_adj xss))) = ⋃# (image_mset mset
(mset xss))
by(induction xss rule: merge_adj.induct) (auto simp: mset_merge)
```

```
lemma mset_merge_all:
  mset (merge_all xss) = (⋃# (mset (map mset xss)))
by(induction xss rule: merge_all.induct) (auto simp: mset_merge mset_merge_adj)
```

lemma *mset_msort_bu*: *mset* (*msort_bu xs*) = *mset xs*
by(*simp add*: *msort_bu_def mset_merge_all comp_def*)

lemma *sorted_merge_adj*:
∀ *xs* ∈ *set xss*. *sorted xs* ⟹ ∀ *xs* ∈ *set* (*merge_adj xss*). *sorted xs*
by(*induction xss rule*: *merge_adj.induct*) (*auto simp*: *sorted_merge*)

lemma *sorted_merge_all*:
∀ *xs* ∈ *set xss*. *sorted xs* ⟹ *sorted* (*merge_all xss*)
apply(*induction xss rule*: *merge_all.induct*)
using [[*simp_depth_limit*=*3*]] **by** (*auto simp add*: *sorted_merge_adj*)

lemma *sorted_msort_bu*: *sorted* (*msort_bu xs*)
by(*simp add*: *msort_bu_def sorted_merge_all*)

1.3.2 Time Complexity

fun *c_merge_adj* :: ('*a*::*linorder*) *list list* ⇒ *nat* **where**
c_merge_adj [] = *0* |
c_merge_adj [*xs*] = *0* |
c_merge_adj (*xs* # *ys* # *zss*) = *c_merge xs ys* + *c_merge_adj zss*

fun *c_merge_all* :: ('*a*::*linorder*) *list list* ⇒ *nat* **where**
c_merge_all [] = *0* |
c_merge_all [*xs*] = *0* |
c_merge_all xss = *c_merge_adj xss* + *c_merge_all* (*merge_adj xss*)

definition *c_msort_bu* :: ('*a*::*linorder*) *list* ⇒ *nat* **where**
c_msort_bu xs = *c_merge_all* (*map* (λ*x*. [*x*]) *xs*)

lemma *length_merge_adj*:
⟦ *even*(*length xss*); ∀ *xs* ∈ *set xss*. *length xs* = *m* ⟧
⟹ ∀ *xs* ∈ *set* (*merge_adj xss*). *length xs* = *2*∗*m*
by(*induction xss rule*: *merge_adj.induct*) (*auto simp*: *length_merge*)

lemma *c_merge_adj*: ∀ *xs* ∈ *set xss*. *length xs* = *m* ⟹ *c_merge_adj xss* ≤ *m* ∗ *length xss*
proof(*induction xss rule*: *c_merge_adj.induct*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *simp*
next
case (*3 x y*) **thus** *?case* **using** *c_merge_ub*[*of x y*] **by** (*simp add*: *alge-*

bra_simps)
qed

lemma *c_merge_all*: ⟦ ∀ *xs* ∈ *set xss*. *length xs* = *m*; *length xss* = *2^k* ⟧
⟹ *c_merge_all xss* ≤ *m* ∗ *k* ∗ *2^k*
proof (*induction xss arbitrary*: *k m rule*: *c_merge_all.induct*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *simp*
next
case (*3 xs ys xss*)
let *?xss* = *xs* # *ys* # *xss*
let *?xss2* = *merge_adj ?xss*
obtain *k'* **where** *k'*: *k* = *Suc k'* **using** *3.prems*(*2*)
by (*metis length_Cons nat.inject nat_power_eq_Suc_0_iff nat.exhaust*)
have *even* (*length ?xss*) **using** *3.prems*(*2*) *k'* **by** *auto*
from *length_merge_adj*[*OF this 3.prems*(*1*)]
have ∗: ∀ *x* ∈ *set*(*merge_adj ?xss*). *length x* = *2*∗*m* **.**
have ∗∗: *length ?xss2* = *2* ^ *k'* **using** *3.prems*(*2*) *k'* **by** *auto*
have *c_merge_all ?xss* = *c_merge_adj ?xss* + *c_merge_all ?xss2* **by** *simp*
also have ... ≤ *m* ∗ *2^k* + *c_merge_all ?xss2*
using *3.prems*(*2*) *c_merge_adj*[*OF 3.prems*(*1*)] **by** (*auto simp*: *algebra_simps*)
also have ... ≤ *m* ∗ *2^k* + (*2*∗*m*) ∗ *k'* ∗ *2^k'*
using *3.IH*[*OF* ∗ ∗∗] **by** *simp*
also have ... = *m* ∗ *k* ∗ *2^k*
using *k'* **by** (*simp add*: *algebra_simps*)
finally show *?case* **.**
qed

corollary *c_msort_bu*: *length xs* = *2* ^ *k* ⟹ *c_msort_bu xs* ≤ *k* ∗ *2* ^ *k*
using *c_merge_all*[*of map* (λ*x*. [*x*]) *xs 1*] **by** (*simp add*: *c_msort_bu_def*)

1.4 Quicksort

fun *quicksort* :: ('*a*::*linorder*) *list* ⇒ '*a list* **where**
quicksort [] = [] |
quicksort (*x*#*xs*) = *quicksort* (*filter* (λ*y*. *y* < *x*) *xs*) @ [*x*] @ *quicksort* (*filter* (λ*y*. *x* ≤ *y*) *xs*)

lemma *mset_quicksort*: *mset* (*quicksort xs*) = *mset xs*
apply (*induction xs rule*: *quicksort.induct*)
apply (*auto simp*: *not_le*)
done

lemma *set_quicksort*: *set* (*quicksort xs*) = *set xs*
by(*rule mset_eq_setD*[*OF mset_quicksort*])

lemma *sorted_quicksort*: *sorted* (*quicksort xs*)
apply (*induction xs rule*: *quicksort.induct*)
apply (*auto simp add*: *sorted_append set_quicksort*)
done

1.5 Insertion Sort w.r.t. Keys and Stability

Note that *insort_key* is already defined in theory *HOL.List*. Thus some of the lemmas are already present as well.

fun *isort_key* :: ('*a* ⇒ '*k*::*linorder*) ⇒ '*a list* ⇒ '*a list* **where**
isort_key f [] = [] |
isort_key f (*x* # *xs*) = *insort_key f x* (*isort_key f xs*)

1.5.1 Standard functional correctness

lemma *mset_insort_key*: *mset* (*insort_key f x xs*) = *add_mset x* (*mset xs*)
by(*induction xs*) *simp_all*

lemma *mset_isort_key*: *mset* (*isort_key f xs*) = *mset xs*
by(*induction xs*) (*simp_all add*: *mset_insort_key*)

lemma *set_isort_key*: *set* (*isort_key f xs*) = *set xs*
by (*rule mset_eq_setD*[*OF mset_isort_key*])

lemma *sorted_insort_key*: *sorted* (*map f* (*insort_key f a xs*)) = *sorted* (*map f xs*)
by(*induction xs*)(*auto simp*: *set_insort_key*)

lemma *sorted_isort_key*: *sorted* (*map f* (*isort_key f xs*))
by(*induction xs*)(*simp_all add*: *sorted_insort_key*)

1.5.2 Stability

lemma *insort_is_Cons*: ∀ *x*∈*set xs*. *f a* ≤ *f x* ⟹ *insort_key f a xs* = *a* # *xs*
by (*cases xs*) *auto*

lemma *filter_insort_key_neg*:
¬ *P x* ⟹ *filter P* (*insort_key f x xs*) = *filter P xs*
by (*induction xs*) *simp_all*

lemma *filter_insort_key_pos*:
sorted (*map f xs*) $\Longrightarrow$ *P x* $\Longrightarrow$ *filter P* (*insort_key f x xs*) = *insort_key f x* (*filter P xs*)
by (*induction xs*) (*auto*, *subst insort_is_Cons*, *auto*)

lemma *sort_key_stable*: *filter* ($\lambda y.$ *f y* = *k*) (*isort_key f xs*) = *filter* ($\lambda y.$ *f y* = *k*) *xs*
proof (*induction xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a xs*)
thus *?case*
proof (*cases f a* = *k*)
case *False* **thus** *?thesis* **by** (*simp add*: *Cons.IH filter_insort_key_neg*)
next
case *True*
have *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*)) **by** *simp*
also have ... = *insort_key f a* (*filter* ($\lambda y.$ *f y* = *k*) (*isort_key f xs*))
by (*simp add*: *True filter_insort_key_pos sorted_isort_key*)
also have ... = *insort_key f a* (*filter* ($\lambda y.$ *f y* = *k*) *xs*) **by** (*simp add*: *Cons.IH*)
also have ... = *a* # (*filter* ($\lambda y.$ *f y* = *k*) *xs*) **by**(*simp add*: *True insort_is_Cons*)
also have ... = *filter* ($\lambda y.$ *f y* = *k*) (*a* # *xs*) **by** (*simp add*: *True*)
finally show *?thesis* **.**
qed
qed

end

2 Creating Balanced Trees

theory *Balance*
imports
HOL−Library.Tree_Real
begin

fun *bal* :: *nat* ⇒ *'a list* ⇒ *'a tree* ∗ *'a list* **where**
bal n xs = (*if n=0 then* (*Leaf*,*xs*) *else*
(*let m* = *n div 2*;
(*l*, *ys*) = *bal m xs*;

```
    (r, zs) = bal (n−1−m) (tl ys)
  in (Node l (hd ys) r, zs)))

declare bal.simps[simp del]

definition bal_list :: nat ⇒ 'a list ⇒ 'a tree where
bal_list n xs = fst (bal n xs)

definition balance_list :: 'a list ⇒ 'a tree where
balance_list xs = bal_list (length xs) xs

definition bal_tree :: nat ⇒ 'a tree ⇒ 'a tree where
bal_tree n t = bal_list n (inorder t)

definition balance_tree :: 'a tree ⇒ 'a tree where
balance_tree t = bal_tree (size t) t

lemma bal_simps:
  bal 0 xs = (Leaf, xs)
  n > 0 ⟹
   bal n xs =
  (let m = n div 2;
      (l, ys) = bal m xs;
      (r, zs) = bal (n−1−m) (tl ys)
  in (Node l (hd ys) r, zs))
by(simp_all add: bal.simps)
```

Some of the following lemmas take advantage of the fact that *bal xs n* yields a result even if $n > length\ xs$.

```
lemma size_bal: bal n xs = (t,ys) ⟹ size t = n
proof(induction n xs arbitrary: t ys rule: bal.induct)
  case (1 n xs)
  thus ?case
    by(cases n=0)
      (auto simp add: bal_simps Let_def split: prod.splits)
qed

lemma bal_inorder:
  ⟦ bal n xs = (t,ys); n ≤ length xs ⟧
  ⟹ inorder t = take n xs ∧ ys = drop n xs
proof(induction n xs arbitrary: t ys rule: bal.induct)
  case (1 n xs) show ?case
  proof cases
    assume n = 0 thus ?thesis using 1 by (simp add: bal_simps)
```

```
  next
    assume [arith]: n ≠ 0
    let ?n1 = n div 2 let ?n2 = n − 1 − ?n1
    from 1.prems obtain l r xs′ where
      b1: bal ?n1 xs = (l,xs′) and
      b2: bal ?n2 (tl xs′) = (r,ys) and
      t: t = ⟨l, hd xs′, r⟩
      by(auto simp: Let_def bal_simps split: prod.splits)
    have IH1: inorder l = take ?n1 xs ∧ xs′ = drop ?n1 xs
      using b1 1.prems by(intro 1.IH(1)) auto
    have IH2: inorder r = take ?n2 (tl xs′) ∧ ys = drop ?n2 (tl xs′)
      using b1 b2 IH1 1.prems by(intro 1.IH(2)) auto
    have drop (n div 2) xs ≠ [] using 1.prems(2) by simp
    hence hd (drop ?n1 xs) # take ?n2 (tl (drop ?n1 xs)) = take (?n2 +
1) (drop ?n1 xs)
      by (metis Suc_eq_plus1 take_Suc)
    hence ∗: inorder t = take n xs using t IH1 IH2
      using take_add[of ?n1 ?n2+1 xs] by(simp)
    have n − n div 2 + n div 2 = n by simp
   hence ys = drop n xs using IH1 IH2 by (simp add: drop_Suc[symmetric])
    thus ?thesis using ∗ by blast
  qed
qed

corollary inorder_bal_list[simp]:
  n ≤ length xs ⟹ inorder(bal_list n xs) = take n xs
unfolding bal_list_def by (metis bal_inorder eq_fst_iff)

corollary inorder_balance_list[simp]: inorder(balance_list xs) = xs
by(simp add: balance_list_def)

corollary inorder_bal_tree:
  n ≤ size t ⟹ inorder(bal_tree n t) = take n (inorder t)
by(simp add: bal_tree_def)

corollary inorder_balance_tree[simp]: inorder(balance_tree t) = inorder t
by(simp add: balance_tree_def inorder_bal_tree)

corollary size_bal_list[simp]: size(bal_list n xs) = n
unfolding bal_list_def by (metis prod.collapse size_bal)

corollary size_balance_list[simp]: size(balance_list xs) = length xs
by (simp add: balance_list_def)
```

corollary *size_bal_tree*[*simp*]: *size*(*bal_tree n t*) = *n*
by(*simp add*: *bal_tree_def*)

corollary *size_balance_tree*[*simp*]: *size*(*balance_tree t*) = *size t*
by(*simp add*: *balance_tree_def*)

lemma *min_height_bal*:
  bal n xs = (*t*,*ys*) ⟹ *min_height t* = *nat*(⌊*log 2* (*n* + *1*)⌋)
proof(*induction n xs arbitrary*: *t ys rule*: *bal.induct*)
  case (*1 n xs*) **show** *?case*
  proof *cases*
    assume *n* = *0* **thus** *?thesis*
      using *1.prems* **by** (*simp add*: *bal_simps*)
  next
    assume [*arith*]: *n* ≠ *0*
    from *1.prems* **obtain** *l r xs'* **where**
      b1: *bal* (*n div 2*) *xs* = (*l*,*xs'*) **and**
      b2: *bal* (*n* − *1* − *n div 2*) (*tl xs'*) = (*r*,*ys*) **and**
      t: *t* = ⟨*l*, *hd xs'*, *r*⟩
      by(*auto simp*: *bal_simps Let_def split*: *prod.splits*)
    let *?log1* = *nat* (*floor*(*log 2* (*n div 2* + *1*)))
    let *?log2* = *nat* (*floor*(*log 2* (*n* − *1* − *n div 2* + *1*)))
    have *IH1*: *min_height l* = *?log1* **using** *1.IH*(*1*) *b1* **by** *simp*
    have *IH2*: *min_height r* = *?log2* **using** *1.IH*(*2*) *b1 b2* **by** *simp*
    have (*n*+*1*) *div 2* ≥ *1* **by** *arith*
    hence *0*: *log 2* ((*n*+*1*) *div 2*) ≥ *0* **by** *simp*
    have *n* − *1* − *n div 2* + *1* ≤ *n div 2* + *1* **by** *arith*
    hence *le*: *?log2* ≤ *?log1*
      by(*simp add*: *nat_mono floor_mono*)
    have *min_height t* = *min ?log1 ?log2* + *1* **by** (*simp add*: *t IH1 IH2*)
    also have ... = *?log2* + *1* **using** *le* **by** (*simp add*: *min_absorb2*)
    also have *n* − *1* − *n div 2* + *1* = (*n*+*1*) *div 2* **by** *linarith*
    also have *nat* (*floor*(*log 2* ((*n*+*1*) *div 2*))) + *1*
      = *nat* (*floor*(*log 2* ((*n*+*1*) *div 2*) + *1*))
      using *0* **by** *linarith*
    also have ... = *nat* (*floor*(*log 2* (*n* + *1*)))
      using *floor_log2_div2*[*of n*+*1*] **by** (*simp add*: *log_mult*)
    finally show *?thesis* **.**
  qed
qed

lemma *height_bal*:
  bal n xs = (*t*,*ys*) ⟹ *height t* = *nat* ⌈*log 2* (*n* + *1*)⌉
proof(*induction n xs arbitrary*: *t ys rule*: *bal.induct*)

```
  case (1 n xs) show ?case
  proof cases
    assume n = 0 thus ?thesis
      using 1.prems by (simp add: bal_simps)
  next
    assume [arith]: n ≠ 0
    from 1.prems obtain l r xs' where
      b1: bal (n div 2) xs = (l,xs') and
      b2: bal (n − 1 − n div 2) (tl xs') = (r,ys) and
      t: t = ⟨l, hd xs', r⟩
      by(auto simp: bal_simps Let_def split: prod.splits)
    let ?log1 = nat ⌈log 2 (n div 2 + 1)⌉
    let ?log2 = nat ⌈log 2 (n − 1 − n div 2 + 1)⌉
    have IH1: height l = ?log1 using 1.IH(1) b1 by simp
    have IH2: height r = ?log2 using 1.IH(2) b1 b2 by simp
    have 0: log 2 (n div 2 + 1) ≥ 0 by auto
    have n − 1 − n div 2 + 1 ≤ n div 2 + 1 by arith
    hence le: ?log2 ≤ ?log1
      by(simp add: nat_mono ceiling_mono del: nat_ceiling_le_eq)
    have height t = max ?log1 ?log2 + 1 by (simp add: t IH1 IH2)
    also have ... = ?log1 + 1 using le by (simp add: max_absorb1)
    also have ... = nat ⌈log 2 (n div 2 + 1) + 1⌉ using 0 by linarith
    also have ... = nat ⌈log 2 (n + 1)⌉
      using ceiling_log2_div2[of n+1] by (simp)
    finally show ?thesis .
  qed
qed

lemma balanced_bal:
  assumes bal n xs = (t,ys) shows balanced t
unfolding balanced_def
using height_bal[OF assms] min_height_bal[OF assms]
by linarith

lemma height_bal_list:
  n ≤ length xs ⟹ height (bal_list n xs) = nat ⌈log 2 (n + 1)⌉
unfolding bal_list_def by (metis height_bal prod.collapse)

lemma height_balance_list:
  height (balance_list xs) = nat ⌈log 2 (length xs + 1)⌉
by (simp add: balance_list_def height_bal_list)

corollary height_bal_tree:
  n ≤ length xs ⟹ height (bal_tree n t) = nat⌈log 2 (n + 1)⌉
```

unfolding *bal_list_def bal_tree_def*
using *height_bal prod.exhaust_sel* **by** *blast*

corollary *height_balance_tree*:
height (*balance_tree t*) = *nat*⌈*log 2* (*size t* + *1*)⌉
by (*simp add*: *bal_tree_def balance_tree_def height_bal_list*)

corollary *balanced_bal_list*[*simp*]: *balanced* (*bal_list n xs*)
unfolding *bal_list_def* **by** (*metis balanced_bal prod.collapse*)

corollary *balanced_balance_list*[*simp*]: *balanced* (*balance_list xs*)
by (*simp add*: *balance_list_def*)

corollary *balanced_bal_tree*[*simp*]: *balanced* (*bal_tree n t*)
by (*simp add*: *bal_tree_def*)

corollary *balanced_balance_tree*[*simp*]: *balanced* (*balance_tree t*)
by (*simp add*: *balance_tree_def*)

lemma *wbalanced_bal*: *bal n xs* = (*t*,*ys*) ⟹ *wbalanced t*
proof(*induction n xs arbitrary*: *t ys rule*: *bal.induct*)
case (*1 n xs*)
show *?case*
proof *cases*
assume *n* = *0*
thus *?thesis*
using *1.prems* **by**(*simp add*: *bal_simps*)
next
assume *n* ≠ *0*
with *1.prems* **obtain** *l ys r zs* **where**
rec1: *bal* (*n div 2*) *xs* = (*l*, *ys*) **and**
rec2: *bal* (*n* − *1* − *n div 2*) (*tl ys*) = (*r*, *zs*) **and**
t: *t* = ⟨*l*, *hd ys*, *r*⟩
by(*auto simp add*: *bal_simps Let_def split*: *prod.splits*)
have *l*: *wbalanced l* **using** *1.IH*(*1*)[*OF* ‹*n*≠*0*› *refl rec1*] **.**
have *wbalanced r* **using** *1.IH*(*2*)[*OF* ‹*n*≠*0*› *refl rec1*[*symmetric*] *refl rec2*] **.**
with *l t size_bal*[*OF rec1*] *size_bal*[*OF rec2*]
show *?thesis* **by** *auto*
qed
qed

An alternative proof via *wbalanced ?t* ⟹ *balanced ?t*:

lemma *bal n xs* = (*t*,*ys*) ⟹ *balanced t*

by(*rule balanced_if_wbalanced*[*OF wbalanced_bal*])

lemma *wbalanced_bal_list*[*simp*]: *wbalanced* (*bal_list n xs*)
by(*simp add*: *bal_list_def*) (*metis prod.collapse wbalanced_bal*)

lemma *wbalanced_balance_list*[*simp*]: *wbalanced* (*balance_list xs*)
by(*simp add*: *balance_list_def*)

lemma *wbalanced_bal_tree*[*simp*]: *wbalanced* (*bal_tree n t*)
by(*simp add*: *bal_tree_def*)

lemma *wbalanced_balance_tree*: *wbalanced* (*balance_tree t*)
by (*simp add*: *balance_tree_def*)

hide_const (**open**) *bal*

end

3 Three-Way Comparison

theory *Cmp*
imports *Main*
begin

datatype *cmp_val* = *LT* | *EQ* | *GT*

definition *cmp* :: *'a*:: *linorder* ⇒ *'a* ⇒ *cmp_val* **where**
cmp x y = (*if x* < *y then LT else if x*=*y then EQ else GT*)

lemma
  LT[*simp*]: *cmp x y* = *LT* ⟷ *x* < *y*
and *EQ*[*simp*]: *cmp x y* = *EQ* ⟷ *x* = *y*
and *GT*[*simp*]: *cmp x y* = *GT* ⟷ *x* > *y*
by (*auto simp*: *cmp_def*)

lemma *case_cmp_if*[*simp*]: (*case c of EQ* ⇒ *e* | *LT* ⇒ *l* | *GT* ⇒ *g*) =
  (*if c* = *LT then l else if c* = *GT then g else e*)
by(*simp split*: *cmp_val.split*)

end

4 Lists Sorted wrt <

theory *Sorted_Less*
imports *Less_False*
begin

hide_const *sorted*

Is a list sorted without duplicates, i.e., wrt <?.

abbreviation *sorted* :: *'a::linorder list* ⇒ *bool* **where**
sorted ≡ *sorted_wrt* (<)

lemmas *sorted_wrt_Cons* = *sorted_wrt.simps*(*2*)

The definition of *sorted_wrt* relates each element to all the elements after it. This causes a blowup of the formulas. Thus we simplify matters by only comparing adjacent elements.

declare
sorted_wrt.simps(*2*)[*simp del*]
sorted_wrt1[*simp*] *sorted_wrt2*[*OF transp_less*, *simp*]

lemma *sorted_cons*: *sorted* (*x*#*xs*) ⟹ *sorted xs*
by(*simp add*: *sorted_wrt_Cons*)

lemma *sorted_cons'*: *ASSUMPTION* (*sorted* (*x*#*xs*)) ⟹ *sorted xs*
by(*rule ASSUMPTION_D* [*THEN sorted_cons*])

lemma *sorted_snoc*: *sorted* (*xs* @ [*y*]) ⟹ *sorted xs*
by(*simp add*: *sorted_wrt_append*)

lemma *sorted_snoc'*: *ASSUMPTION* (*sorted* (*xs* @ [*y*])) ⟹ *sorted xs*
by(*rule ASSUMPTION_D* [*THEN sorted_snoc*])

lemma *sorted_mid_iff*:
sorted(*xs* @ *y* # *ys*) = (*sorted*(*xs* @ [*y*]) ∧ *sorted*(*y* # *ys*))
by(*fastforce simp add*: *sorted_wrt_Cons sorted_wrt_append*)

lemma *sorted_mid_iff2*:
sorted(*x* # *xs* @ *y* # *ys*) =
(*sorted*(*x* # *xs*) ∧ *x* < *y* ∧ *sorted*(*xs* @ [*y*]) ∧ *sorted*(*y* # *ys*))
by(*fastforce simp add*: *sorted_wrt_Cons sorted_wrt_append*)

lemma *sorted_mid_iff'*: *NO_MATCH* [] *ys* ⟹
sorted(*xs* @ *y* # *ys*) = (*sorted*(*xs* @ [*y*]) ∧ *sorted*(*y* # *ys*))

by(*rule sorted_mid_iff*)

lemmas *sorted_lems* = *sorted_mid_iff' sorted_mid_iff2 sorted_cons' sorted_snoc'*

Splay trees need two additional *sorted* lemmas:

lemma *sorted_snoc_le*:
ASSUMPTION(*sorted*(*xs* @ [*x*])) ⟹ *x* ≤ *y* ⟹ *sorted* (*xs* @ [*y*])
by (*auto simp add*: *sorted_wrt_append ASSUMPTION_def*)

lemma *sorted_Cons_le*:
ASSUMPTION(*sorted*(*x* # *xs*)) ⟹ *y* ≤ *x* ⟹ *sorted* (*y* # *xs*)
by (*auto simp add*: *sorted_wrt_Cons ASSUMPTION_def*)

end

5 List Insertion and Deletion

theory *List_Ins_Del*
imports *Sorted_Less*
begin

5.1 Elements in a list

lemma *sorted_Cons_iff*:
sorted(*x* # *xs*) = ((∀ *y* ∈ *set xs*. *x* < *y*) ∧ *sorted xs*)
by(*simp add*: *sorted_wrt_Cons*)

lemma *sorted_snoc_iff*:
sorted(*xs* @ [*x*]) = (*sorted xs* ∧ (∀ *y* ∈ *set xs*. *y* < *x*))
by(*simp add*: *sorted_wrt_append*)

lemmas *isin_simps* = *sorted_lems sorted_Cons_iff sorted_snoc_iff*

5.2 Inserting into an ordered list without duplicates:

fun *ins_list* :: *'a*::*linorder* ⇒ *'a list* ⇒ *'a list* **where**
ins_list x [] = [*x*] |
ins_list x (*a*#*xs*) =
(*if x* < *a then x*#*a*#*xs else if x*=*a then a*#*xs else a* # *ins_list x xs*)

lemma *set_ins_list*: *set* (*ins_list x xs*) = *insert x* (*set xs*)
by(*induction xs*) *auto*

lemma *distinct_if_sorted*: *sorted xs* $\Longrightarrow$ *distinct xs*
apply(*induction xs rule*: *induct_list012*)
apply *auto*
by (*metis in_set_conv_decomp_first less_imp_not_less sorted_mid_iff2*)

lemma *sorted_ins_list*: *sorted xs* $\Longrightarrow$ *sorted*(*ins_list x xs*)
by(*induction xs rule*: *induct_list012*) *auto*

lemma *ins_list_sorted*: *sorted* (*xs* @ [*a*]) $\Longrightarrow$
ins_list x (*xs* @ *a* # *ys*) =
(*if x* < *a then ins_list x xs* @ (*a*#*ys*) *else xs* @ *ins_list x* (*a*#*ys*))
by(*induction xs*) (*auto simp*: *sorted_lems*)

In principle, *sorted* (*?xs* @ [*?a*]) $\Longrightarrow$ *ins_list ?x* (*?xs* @ *?a* # *?ys*) = (*if ?x* < *?a then ins_list ?x ?xs* @ *?a* # *?ys else ?xs* @ *ins_list ?x* (*?a* # *?ys*)) suffices, but the following two corollaries speed up proofs.

corollary *ins_list_sorted1*: *sorted* (*xs* @ [*a*]) $\Longrightarrow$ *a* $\leq$ *x* $\Longrightarrow$
ins_list x (*xs* @ *a* # *ys*) = *xs* @ *ins_list x* (*a*#*ys*)
by(*auto simp add*: *ins_list_sorted*)

corollary *ins_list_sorted2*: *sorted* (*xs* @ [*a*]) $\Longrightarrow$ *x* < *a* $\Longrightarrow$
ins_list x (*xs* @ *a* # *ys*) = *ins_list x xs* @ (*a*#*ys*)
by(*auto simp*: *ins_list_sorted*)

lemmas *ins_list_simps* = *sorted_lems ins_list_sorted1 ins_list_sorted2*

Splay trees need two additional *ins_list* lemmas:

lemma *ins_list_Cons*: *sorted* (*x* # *xs*) $\Longrightarrow$ *ins_list x xs* = *x* # *xs*
by (*induction xs*) *auto*

lemma *ins_list_snoc*: *sorted* (*xs* @ [*x*]) $\Longrightarrow$ *ins_list x xs* = *xs* @ [*x*]
by(*induction xs*) (*auto simp add*: *sorted_mid_iff2*)

5.3 Delete one occurrence of an element from a list:

fun *del_list* :: *'a* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**
del_list x [] = [] |
del_list x (*a*#*xs*) = (*if x*=*a then xs else a* # *del_list x xs*)

lemma *del_list_idem*: *x* $\notin$ *set xs* $\Longrightarrow$ *del_list x xs* = *xs*
by (*induct xs*) *simp_all*

lemma *set_del_list_eq*:
distinct xs $\Longrightarrow$ *set* (*del_list x xs*) = *set xs* − {*x*}

by(*induct xs*) *auto*

lemma *sorted_del_list*: *sorted xs* $\Longrightarrow$ *sorted*(*del_list x xs*)
apply(*induction xs rule*: *induct_list012*)
apply *auto*
by (*meson order.strict_trans sorted_Cons_iff*)

lemma *del_list_sorted*: *sorted* (*xs* @ *a* # *ys*) $\Longrightarrow$
del_list x (*xs* @ *a* # *ys*) = (*if x* < *a then del_list x xs* @ *a* # *ys else xs* @ *del_list x* (*a* # *ys*))
by(*induction xs*)
(*fastforce simp*: *sorted_lems sorted_Cons_iff intro*!: *del_list_idem*)+

In principle, *sorted* (*?xs* @ *?a* # *?ys*) $\Longrightarrow$ *del_list ?x* (*?xs* @ *?a* # *?ys*) = (*if ?x* < *?a then del_list ?x ?xs* @ *?a* # *?ys else ?xs* @ *del_list ?x* (*?a* # *?ys*)) suffices, but the following corollaries speed up proofs.

corollary *del_list_sorted1*: *sorted* (*xs* @ *a* # *ys*) $\Longrightarrow$ *a* $\leq$ *x* $\Longrightarrow$
del_list x (*xs* @ *a* # *ys*) = *xs* @ *del_list x* (*a* # *ys*)
by (*auto simp*: *del_list_sorted*)

corollary *del_list_sorted2*: *sorted* (*xs* @ *a* # *ys*) $\Longrightarrow$ *x* < *a* $\Longrightarrow$
del_list x (*xs* @ *a* # *ys*) = *del_list x xs* @ *a* # *ys*
by (*auto simp*: *del_list_sorted*)

corollary *del_list_sorted3*:
sorted (*xs* @ *a* # *ys* @ *b* # *zs*) $\Longrightarrow$ *x* < *b* $\Longrightarrow$
del_list x (*xs* @ *a* # *ys* @ *b* # *zs*) = *del_list x* (*xs* @ *a* # *ys*) @ *b* # *zs*
by (*auto simp*: *del_list_sorted sorted_lems*)

corollary *del_list_sorted4*:
sorted (*xs* @ *a* # *ys* @ *b* # *zs* @ *c* # *us*) $\Longrightarrow$ *x* < *c* $\Longrightarrow$
del_list x (*xs* @ *a* # *ys* @ *b* # *zs* @ *c* # *us*) = *del_list x* (*xs* @ *a* # *ys* @ *b* # *zs*) @ *c* # *us*
by (*auto simp*: *del_list_sorted sorted_lems*)

corollary *del_list_sorted5*:
sorted (*xs* @ *a* # *ys* @ *b* # *zs* @ *c* # *us* @ *d* # *vs*) $\Longrightarrow$ *x* < *d* $\Longrightarrow$
del_list x (*xs* @ *a* # *ys* @ *b* # *zs* @ *c* # *us* @ *d* # *vs*) =
del_list x (*xs* @ *a* # *ys* @ *b* # *zs* @ *c* # *us*) @ *d* # *vs*
by (*auto simp*: *del_list_sorted sorted_lems*)

lemmas *del_list_simps* = *sorted_lems*
del_list_sorted1
del_list_sorted2

del_list_sorted3
del_list_sorted4
del_list_sorted5

Splay trees need two additional *del_list* lemmas:

lemma *del_list_notin_Cons*: *sorted* $(x \# xs) \Longrightarrow$ *del_list* $x\ xs = xs$
by(*induction xs*)(*fastforce simp*: *sorted_Cons_iff*)+

lemma *del_list_sorted_app*:
$sorted(xs\ @\ [x]) \Longrightarrow$ *del_list* $x\ (xs\ @\ ys) = xs\ @$ *del_list* $x\ ys$
by (*induction xs*) (*auto simp*: *sorted_mid_iff2*)

end

6 Specifications of Set ADT

theory *Set_Specs*
imports *List_Ins_Del*
begin

The basic set interface with traditional *set*-based specification:

locale *Set* =
fixes *empty* :: $'s$
fixes *insert* :: $'a \Rightarrow 's \Rightarrow 's$
fixes *delete* :: $'a \Rightarrow 's \Rightarrow 's$
fixes *isin* :: $'s \Rightarrow 'a \Rightarrow$ *bool*
fixes *set* :: $'s \Rightarrow 'a$ *set*
fixes *invar* :: $'s \Rightarrow$ *bool*
assumes *set_empty*: $set\ empty = \{\}$
assumes *set_isin*: $invar\ s \Longrightarrow isin\ s\ x = (x \in set\ s)$
assumes *set_insert*: $invar\ s \Longrightarrow set(insert\ x\ s) = Set.insert\ x\ (set\ s)$
assumes *set_delete*: $invar\ s \Longrightarrow set(delete\ x\ s) = set\ s - \{x\}$
assumes *invar_empty*: *invar empty*
assumes *invar_insert*: $invar\ s \Longrightarrow invar(insert\ x\ s)$
assumes *invar_delete*: $invar\ s \Longrightarrow invar(delete\ x\ s)$

lemmas (**in** *Set*) *set_specs* =
set_empty set_isin set_insert set_delete invar_empty invar_insert invar_delete

The basic set interface with *inorder*-based specification:

locale *Set_by_Ordered* =
fixes *empty* :: $'t$
fixes *insert* :: $'a$::*linorder* $\Rightarrow 't \Rightarrow 't$
fixes *delete* :: $'a \Rightarrow 't \Rightarrow 't$

fixes *isin* :: $'t \Rightarrow 'a \Rightarrow bool$
fixes *inorder* :: $'t \Rightarrow 'a\ list$
fixes *inv* :: $'t \Rightarrow bool$
assumes *inorder_empty*: $inorder\ empty = []$
assumes *isin*: $inv\ t \wedge sorted(inorder\ t) \Longrightarrow$
$isin\ t\ x = (x \in set\ (inorder\ t))$
assumes *inorder_insert*: $inv\ t \wedge sorted(inorder\ t) \Longrightarrow$
$inorder(insert\ x\ t) = ins\_list\ x\ (inorder\ t)$
assumes *inorder_delete*: $inv\ t \wedge sorted(inorder\ t) \Longrightarrow$
$inorder(delete\ x\ t) = del\_list\ x\ (inorder\ t)$
assumes *inorder_inv_empty*: $inv\ empty$
assumes *inorder_inv_insert*: $inv\ t \wedge sorted(inorder\ t) \Longrightarrow inv(insert\ x\ t)$
assumes *inorder_inv_delete*: $inv\ t \wedge sorted(inorder\ t) \Longrightarrow inv(delete\ x\ t)$

begin

It implements the traditional specification:

definition *set* :: $'t \Rightarrow 'a\ set$ **where**
$set == List.set\ o\ inorder$

definition *invar* :: $'t \Rightarrow bool$ **where**
$invar\ t == inv\ t \wedge sorted\ (inorder\ t)$

sublocale *Set*
empty insert delete isin set invar
proof(*standard*, *goal_cases*)
case *1* **show** *?case* **by** (*auto simp*: *inorder_empty set_def*)
next
case *2* **thus** *?case* **by**(*simp add*: *isin invar_def set_def*)
next
case *3* **thus** *?case* **by**(*simp add*: *inorder_insert set_ins_list set_def invar_def*)
next
case (*4 s x*) **thus** *?case*
by (*auto simp*: *inorder_delete distinct_if_sorted set_del_list_eq invar_def set_def*)
next
case *5* **thus** *?case* **by**(*simp add*: *inorder_empty inorder_inv_empty invar_def*)
next
case *6* **thus** *?case* **by**(*simp add*: *inorder_insert inorder_inv_insert sorted_ins_list invar_def*)
next
case *7* **thus** *?case* **by** (*auto simp*: *inorder_delete inorder_inv_delete sorted_del_list*

invar_def)
qed

end

Set2 = Set with binary operations:

locale *Set2* = *Set*
where *insert* = *insert* **for** *insert* :: *'a* ⇒ *'s* ⇒ *'s* +
fixes *union* :: *'s* ⇒ *'s* ⇒ *'s*
fixes *inter* :: *'s* ⇒ *'s* ⇒ *'s*
fixes *diff* :: *'s* ⇒ *'s* ⇒ *'s*
assumes *set_union*: ⟦ *invar s1*; *invar s2* ⟧ ⟹ *set*(*union s1 s2*) = *set s1* ∪ *set s2*
assumes *set_inter*: ⟦ *invar s1*; *invar s2* ⟧ ⟹ *set*(*inter s1 s2*) = *set s1* ∩ *set s2*
assumes *set_diff*: ⟦ *invar s1*; *invar s2* ⟧ ⟹ *set*(*diff s1 s2*) = *set s1* − *set s2*
assumes *invar_union*: ⟦ *invar s1*; *invar s2* ⟧ ⟹ *invar*(*union s1 s2*)
assumes *invar_inter*: ⟦ *invar s1*; *invar s2* ⟧ ⟹ *invar*(*inter s1 s2*)
assumes *invar_diff*: ⟦ *invar s1*; *invar s2* ⟧ ⟹ *invar*(*diff s1 s2*)

end

7 Unbalanced Tree Implementation of Set

theory *Tree_Set*
imports
HOL−Library.Tree
Cmp
Set_Specs
begin

definition *empty* :: *'a tree* **where**
empty == *Leaf*

fun *isin* :: *'a*::*linorder tree* ⇒ *'a* ⇒ *bool* **where**
isin Leaf x = *False* |
isin (*Node l a r*) *x* =
(*case cmp x a of*
LT ⇒ *isin l x* |
EQ ⇒ *True* |
GT ⇒ *isin r x*)

hide_const (**open**) *insert*

fun *insert* :: *'a::linorder* ⇒ *'a tree* ⇒ *'a tree* **where**
insert x Leaf = *Node Leaf x Leaf* |
insert x (*Node l a r*) =
(*case cmp x a of*
LT ⇒ *Node* (*insert x l*) *a r* |
EQ ⇒ *Node l a r* |
GT ⇒ *Node l a* (*insert x r*))

fun *split_min* :: *'a tree* ⇒ *'a* ∗ *'a tree* **where**
split_min (*Node l a r*) =
(*if l* = *Leaf then* (*a*,*r*) *else let* (*x*,*l'*) = *split_min l in* (*x*, *Node l' a r*))

fun *delete* :: *'a::linorder* ⇒ *'a tree* ⇒ *'a tree* **where**
delete x Leaf = *Leaf* |
delete x (*Node l a r*) =
(*case cmp x a of*
LT ⇒ *Node* (*delete x l*) *a r* |
GT ⇒ *Node l a* (*delete x r*) |
EQ ⇒ *if r* = *Leaf then l else let* (*a'*,*r'*) = *split_min r in Node l a' r'*)

7.1 Functional Correctness Proofs

lemma *isin_set*: *sorted*(*inorder t*) ⟹ *isin t x* = (*x* ∈ *set* (*inorder t*))
by (*induction t*) (*auto simp*: *isin_simps*)

lemma *inorder_insert*:
sorted(*inorder t*) ⟹ *inorder*(*insert x t*) = *ins_list x* (*inorder t*)
by(*induction t*) (*auto simp*: *ins_list_simps*)

lemma *split_minD*:
split_min t = (*x*,*t'*) ⟹ *t* ≠ *Leaf* ⟹ *x* # *inorder t'* = *inorder t*
by(*induction t arbitrary*: *t' rule*: *split_min.induct*)
(*auto simp*: *sorted_lems split*: *prod.splits if_splits*)

lemma *inorder_delete*:
sorted(*inorder t*) ⟹ *inorder*(*delete x t*) = *del_list x* (*inorder t*)
by(*induction t*) (*auto simp*: *del_list_simps split_minD split*: *prod.splits*)

interpretation *S*: *Set_by_Ordered*
where *empty* = *empty* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = λ_. *True*

proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** (*simp add*: *empty_def*)
next
  case *2* **thus** *?case* **by**(*simp add*: *isin_set*)
next
  case *3* **thus** *?case* **by**(*simp add*: *inorder_insert*)
next
  case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
qed (*rule TrueI*)+

end

8 Association List Update and Deletion

theory *AList_Upd_Del*
imports *Sorted_Less*
begin

abbreviation *sorted1 ps* ≡ *sorted*(*map fst ps*)

Define own *map_of* function to avoid pulling in an unknown amount of lemmas implicitly (via the simpset).

hide_const (**open**) *map_of*

fun *map_of* :: ('*a*∗'*b*)*list* ⇒ '*a* ⇒ '*b option* **where**
map_of [] = (λ*x*. *None*) |
map_of ((*a*,*b*)#*ps*) = (λ*x*. *if x*=*a then Some b else map_of ps x*)

Updating an association list:

fun *upd_list* :: '*a*::*linorder* ⇒ '*b* ⇒ ('*a*∗'*b*) *list* ⇒ ('*a*∗'*b*) *list* **where**
upd_list x y [] = [(*x*,*y*)] |
upd_list x y ((*a*,*b*)#*ps*) =
  (*if x* < *a then* (*x*,*y*)#(*a*,*b*)#*ps else*
  if x = *a then* (*x*,*y*)#*ps else* (*a*,*b*) # *upd_list x y ps*)

fun *del_list* :: '*a*::*linorder* ⇒ ('*a*∗'*b*)*list* ⇒ ('*a*∗'*b*)*list* **where**
del_list x [] = [] |
del_list x ((*a*,*b*)#*ps*) = (*if x* = *a then ps else* (*a*,*b*) # *del_list x ps*)

8.1 Lemmas for *map_of*

lemma *map_of_ins_list*: *map_of* (*upd_list x y ps*) = (*map_of ps*)(*x* := *Some y*)
by(*induction ps*) *auto*

lemma *map_of_append*: *map_of* (*ps* @ *qs*) *x* =
(*case map_of ps x of None* ⇒ *map_of qs x* | *Some y* ⇒ *Some y*)
by(*induction ps*)(*auto*)

lemma *map_of_None*: *sorted* (*x* # *map fst ps*) ⟹ *map_of ps x* = *None*
by (*induction ps*) (*fastforce simp*: *sorted_lems sorted_wrt_Cons*)+

lemma *map_of_None2*: *sorted* (*map fst ps* @ [*x*]) ⟹ *map_of ps x* = *None*
by (*induction ps*) (*auto simp*: *sorted_lems*)

lemma *map_of_del_list*: *sorted1 ps* ⟹
map_of(*del_list x ps*) = (*map_of ps*)(*x* := *None*)
by(*induction ps*) (*auto simp*: *map_of_None sorted_lems fun_eq_iff*)

lemma *map_of_sorted_Cons*: *sorted* (*a* # *map fst ps*) ⟹ *x* < *a* ⟹
map_of ps x = *None*
by (*simp add*: *map_of_None sorted_Cons_le*)

lemma *map_of_sorted_snoc*: *sorted* (*map fst ps* @ [*a*]) ⟹ *a* ≤ *x* ⟹
map_of ps x = *None*
by (*simp add*: *map_of_None2 sorted_snoc_le*)

lemmas *map_of_sorteds* = *map_of_sorted_Cons map_of_sorted_snoc*
lemmas *map_of_simps* = *sorted_lems map_of_append map_of_sorteds*

8.2 Lemmas for *upd_list*

lemma *sorted_upd_list*: *sorted1 ps* ⟹ *sorted1* (*upd_list x y ps*)
apply(*induction ps*)
apply *simp*
apply(*case_tac ps*)
apply *auto*
done

lemma *upd_list_sorted*: *sorted1* (*ps* @ [(*a*,*b*)]) ⟹
upd_list x y (*ps* @ (*a*,*b*) # *qs*) =
(*if x* < *a then upd_list x y ps* @ (*a*,*b*) # *qs*
else ps @ *upd_list x y* ((*a*,*b*) # *qs*))
by(*induction ps*) (*auto simp*: *sorted_lems*)

In principle, *sorted1* (*?ps* @ [(*?a*, *?b*)]) ⟹ *upd_list ?x ?y* (*?ps* @ (*?a*, *?b*) # *?qs*) = (*if ?x* < *?a then upd_list ?x ?y ?ps* @ (*?a*, *?b*) # *?qs else ?ps* @ *upd_list ?x ?y* ((*?a*, *?b*) # *?qs*)) suffices, but the following two corollaries speed up proofs.

corollary *upd_list_sorted1*: ⟦ *sorted* (*map fst ps* @ [*a*]); *x* < *a* ⟧ ⟹
upd_list x y (*ps* @ (*a*,*b*) # *qs*) = *upd_list x y ps* @ (*a*,*b*) # *qs*
by (*auto simp*: *upd_list_sorted*)

corollary *upd_list_sorted2*: ⟦ *sorted* (*map fst ps* @ [*a*]); *a* ≤ *x* ⟧ ⟹
upd_list x y (*ps* @ (*a*,*b*) # *qs*) = *ps* @ *upd_list x y* ((*a*,*b*) # *qs*)
by (*auto simp*: *upd_list_sorted*)

lemmas *upd_list_simps* = *sorted_lems upd_list_sorted1 upd_list_sorted2*

Splay trees need two additional *upd_list* lemmas:

lemma *upd_list_Cons*:
sorted1 ((*x*,*y*) # *xs*) ⟹ *upd_list x y xs* = (*x*,*y*) # *xs*
by (*induction xs*) *auto*

lemma *upd_list_snoc*:
sorted1 (*xs* @ [(*x*,*y*)]) ⟹ *upd_list x y xs* = *xs* @ [(*x*,*y*)]
by(*induction xs*) (*auto simp add*: *sorted_mid_iff2*)

8.3 Lemmas for *del_list*

lemma *sorted_del_list*: *sorted1 ps* ⟹ *sorted1* (*del_list x ps*)
apply(*induction ps*)
apply *simp*
apply(*case_tac ps*)
apply (*auto simp*: *sorted_Cons_le*)
done

lemma *del_list_idem*: *x* ∉ *set*(*map fst xs*) ⟹ *del_list x xs* = *xs*
by (*induct xs*) *auto*

lemma *del_list_sorted*: *sorted1* (*ps* @ (*a*,*b*) # *qs*) ⟹
del_list x (*ps* @ (*a*,*b*) # *qs*) =
(*if x* < *a then del_list x ps* @ (*a*,*b*) # *qs*
else ps @ *del_list x* ((*a*,*b*) # *qs*))
by(*induction ps*)
(*fastforce simp*: *sorted_lems sorted_wrt_Cons intro*!: *del_list_idem*)+

In principle, *sorted1* (*?ps* @ (*?a*, *?b*) # *?qs*) ⟹ *del_list ?x* (*?ps* @ (*?a*, *?b*) # *?qs*) = (*if ?x* < *?a then del_list ?x ?ps* @ (*?a*, *?b*) # *?qs else ?ps* @ *del_list ?x* ((*?a*, *?b*) # *?qs*)) suffices, but the following corollaries speed up proofs.

corollary *del_list_sorted1*: *sorted1* (*xs* @ (*a*,*b*) # *ys*) ⟹ *a* ≤ *x* ⟹
del_list x (*xs* @ (*a*,*b*) # *ys*) = *xs* @ *del_list x* ((*a*,*b*) # *ys*)

by (*auto simp*: *del_list_sorted*)

lemma *del_list_sorted2*: *sorted1* (*xs* @ (*a*,*b*) # *ys*) ⟹ *x* < *a* ⟹
del_list x (*xs* @ (*a*,*b*) # *ys*) = *del_list x xs* @ (*a*,*b*) # *ys*
by (*auto simp*: *del_list_sorted*)

lemma *del_list_sorted3*:
sorted1 (*xs* @ (*a*,*a'*) # *ys* @ (*b*,*b'*) # *zs*) ⟹ *x* < *b* ⟹
del_list x (*xs* @ (*a*,*a'*) # *ys* @ (*b*,*b'*) # *zs*) = *del_list x* (*xs* @ (*a*,*a'*) # *ys*) @ (*b*,*b'*) # *zs*
by (*auto simp*: *del_list_sorted sorted_lems*)

lemma *del_list_sorted4*:
sorted1 (*xs* @ (*a*,*a'*) # *ys* @ (*b*,*b'*) # *zs* @ (*c*,*c'*) # *us*) ⟹ *x* < *c* ⟹
del_list x (*xs* @ (*a*,*a'*) # *ys* @ (*b*,*b'*) # *zs* @ (*c*,*c'*) # *us*) = *del_list x* (*xs* @ (*a*,*a'*) # *ys* @ (*b*,*b'*) # *zs*) @ (*c*,*c'*) # *us*
by (*auto simp*: *del_list_sorted sorted_lems*)

lemma *del_list_sorted5*:
sorted1 (*xs* @ (*a*,*a'*) # *ys* @ (*b*,*b'*) # *zs* @ (*c*,*c'*) # *us* @ (*d*,*d'*) # *vs*) ⟹ *x* < *d* ⟹
del_list x (*xs* @ (*a*,*a'*) # *ys* @ (*b*,*b'*) # *zs* @ (*c*,*c'*) # *us* @ (*d*,*d'*) # *vs*) =
del_list x (*xs* @ (*a*,*a'*) # *ys* @ (*b*,*b'*) # *zs* @ (*c*,*c'*) # *us*) @ (*d*,*d'*) # *vs*
by (*auto simp*: *del_list_sorted sorted_lems*)

lemmas *del_list_simps* = *sorted_lems*
del_list_sorted1
del_list_sorted2
del_list_sorted3
del_list_sorted4
del_list_sorted5

Splay trees need two additional *del_list* lemmas:

lemma *del_list_notin_Cons*: *sorted* (*x* # *map fst xs*) ⟹ *del_list x xs* = *xs*
by(*induction xs*)(*fastforce simp*: *sorted_wrt_Cons*)+

lemma *del_list_sorted_app*:
sorted(*map fst xs* @ [*x*]) ⟹ *del_list x* (*xs* @ *ys*) = *xs* @ *del_list x ys*
by (*induction xs*) (*auto simp*: *sorted_mid_iff2*)

end

9 Specifications of Map ADT

theory *Map_Specs*
imports *AList_Upd_Del*
begin

The basic map interface with traditional *set*-based specification:

locale *Map* =
fixes *empty* :: $'m$
fixes *update* :: $'a \Rightarrow 'b \Rightarrow 'm \Rightarrow 'm$
fixes *delete* :: $'a \Rightarrow 'm \Rightarrow 'm$
fixes *lookup* :: $'m \Rightarrow 'a \Rightarrow 'b$ *option*
fixes *invar* :: $'m \Rightarrow$ *bool*
assumes *map_empty*: *lookup empty* = $(\lambda\_.$ *None*$)$
and *map_update*: *invar m* $\Longrightarrow$ *lookup*$($*update a b m*$)$ = $($*lookup m*$)($*a* := *Some b*$)$
and *map_delete*: *invar m* $\Longrightarrow$ *lookup*$($*delete a m*$)$ = $($*lookup m*$)($*a* := *None*$)$
and *invar_empty*: *invar empty*
and *invar_update*: *invar m* $\Longrightarrow$ *invar*$($*update a b m*$)$
and *invar_delete*: *invar m* $\Longrightarrow$ *invar*$($*delete a m*$)$

lemmas (**in** *Map*) *map_specs* =
map_empty map_update map_delete invar_empty invar_update invar_delete

The basic map interface with *inorder*-based specification:

locale *Map_by_Ordered* =
fixes *empty* :: $'t$
fixes *update* :: $'a$::*linorder* $\Rightarrow 'b \Rightarrow 't \Rightarrow 't$
fixes *delete* :: $'a \Rightarrow 't \Rightarrow 't$
fixes *lookup* :: $'t \Rightarrow 'a \Rightarrow 'b$ *option*
fixes *inorder* :: $'t \Rightarrow ('a * 'b)$ *list*
fixes *inv* :: $'t \Rightarrow$ *bool*
assumes *inorder_empty*: *inorder empty* = []
and *inorder_lookup*: *inv t* $\wedge$ *sorted1* $($*inorder t*$)$ $\Longrightarrow$
lookup t a = *map_of* $($*inorder t*$)$ *a*
and *inorder_update*: *inv t* $\wedge$ *sorted1* $($*inorder t*$)$ $\Longrightarrow$
inorder$($*update a b t*$)$ = *upd_list a b* $($*inorder t*$)$
and *inorder_delete*: *inv t* $\wedge$ *sorted1* $($*inorder t*$)$ $\Longrightarrow$
inorder$($*delete a t*$)$ = *del_list a* $($*inorder t*$)$
and *inorder_inv_empty*: *inv empty*
and *inorder_inv_update*: *inv t* $\wedge$ *sorted1* $($*inorder t*$)$ $\Longrightarrow$ *inv*$($*update a b t*$)$
and *inorder_inv_delete*: *inv t* $\wedge$ *sorted1* $($*inorder t*$)$ $\Longrightarrow$ *inv*$($*delete a t*$)$

begin

It implements the traditional specification:

definition *invar* :: $'t \Rightarrow$ *bool* **where**
invar t == *inv t* $\wedge$ *sorted1* (*inorder t*)

sublocale *Map*
empty update delete lookup invar
proof(*standard*, *goal_cases*)
case *1* **show** *?case* **by** (*auto simp*: *inorder_lookup inorder_empty inorder_inv_empty*)
next
case *2* **thus** *?case*
by(*simp add*: *fun_eq_iff inorder_update inorder_inv_update map_of_ins_list inorder_lookup*
sorted_upd_list invar_def)
next
case *3* **thus** *?case*
by(*simp add*: *fun_eq_iff inorder_delete inorder_inv_delete map_of_del_list inorder_lookup*
sorted_del_list invar_def)
next
case *4* **thus** *?case* **by**(*simp add*: *inorder_empty inorder_inv_empty invar_def*)
next
case *5* **thus** *?case* **by**(*simp add*: *inorder_update inorder_inv_update sorted_upd_list invar_def*)
next
case *6* **thus** *?case* **by** (*auto simp*: *inorder_delete inorder_inv_delete sorted_del_list invar_def*)
qed

end

end

10 Unbalanced Tree Implementation of Map

theory *Tree_Map*
imports
Tree_Set
Map_Specs
begin

fun *lookup* :: ($'a$::*linorder*$*'b$) *tree* $\Rightarrow$ $'a \Rightarrow$ $'b$ *option* **where**

```
lookup Leaf x = None |
lookup (Node l (a,b) r) x =
  (case cmp x a of LT ⇒ lookup l x | GT ⇒ lookup r x | EQ ⇒ Some b)

fun update :: 'a::linorder ⇒ 'b ⇒ ('a*'b) tree ⇒ ('a*'b) tree where
update x y Leaf = Node Leaf (x,y) Leaf |
update x y (Node l (a,b) r) = (case cmp x a of
   LT ⇒ Node (update x y l) (a,b) r |
   EQ ⇒ Node l (x,y) r |
   GT ⇒ Node l (a,b) (update x y r))

fun delete :: 'a::linorder ⇒ ('a*'b) tree ⇒ ('a*'b) tree where
delete x Leaf = Leaf |
delete x (Node l (a,b) r) = (case cmp x a of
  LT ⇒ Node (delete x l) (a,b) r |
  GT ⇒ Node l (a,b) (delete x r) |
  EQ ⇒ if r = Leaf then l else let (ab',r') = split_min r in Node l ab' r')
```

10.1 Functional Correctness Proofs

```
lemma lookup_map_of:
  sorted1(inorder t) ⟹ lookup t x = map_of (inorder t) x
by (induction t) (auto simp: map_of_simps split: option.split)

lemma inorder_update:
  sorted1(inorder t) ⟹ inorder(update a b t) = upd_list a b (inorder t)
by(induction t) (auto simp: upd_list_simps)

lemma inorder_delete:
  sorted1(inorder t) ⟹ inorder(delete x t) = del_list x (inorder t)
by(induction t) (auto simp: del_list_simps split_minD split: prod.splits)

interpretation M: Map_by_Ordered
where empty = empty and lookup = lookup and update = update and
delete = delete
and inorder = inorder and inv = λ_. True
proof (standard, goal_cases)
  case 1 show ?case by (simp add: empty_def)
next
  case 2 thus ?case by(simp add: lookup_map_of)
next
  case 3 thus ?case by(simp add: inorder_update)
next
  case 4 thus ?case by(simp add: inorder_delete)
```

qed *auto*

end
theory *Tree2*
imports *Main*
begin

datatype $('a,'b)$ *tree* =
  Leaf $(\langle\rangle)$ |
  Node $('a,'b)$*tree* $'a$ $'b$ $('a,'b)$ *tree* $((1\langle\_,/\ \_,/\ \_,/\ \_\rangle))$

fun *inorder* :: $('a,'b)$*tree* $\Rightarrow$ $'a$ *list* **where**
inorder Leaf = [] |
inorder (*Node l a* _ *r*) = *inorder l* @ *a* # *inorder r*

fun *height* :: $('a,'b)$ *tree* $\Rightarrow$ *nat* **where**
height Leaf = *0* |
height (*Node l a* _ *r*) = *max* (*height l*) (*height r*) + *1*

fun *set_tree* :: $('a,'b)$ *tree* $\Rightarrow$ $'a$ *set* **where**
set_tree Leaf = {} |
set_tree (*Node l a* _ *r*) = *Set.insert a* (*set_tree l* $\cup$ *set_tree r*)

fun *bst* :: $('a$::*linorder*,$'b)$ *tree* $\Rightarrow$ *bool* **where**
bst Leaf = *True* |
bst (*Node l a* _ *r*) = (*bst l* $\wedge$ *bst r* $\wedge$ ($\forall x \in$ *set_tree l*. *x* < *a*) $\wedge$ ($\forall x \in$ *set_tree r*. *a* < *x*))

fun *size1* :: $('a,'b)$ *tree* $\Rightarrow$ *nat* **where**
size1 $\langle\rangle$ = *1* |
size1 $\langle l, \_, \_, r\rangle$ = *size1 l* + *size1 r*

lemma *size1_size*: *size1 t* = *size t* + *1*
by (*induction t*) *simp_all*

lemma *size1_ge0*[*simp*]: *0* < *size1 t*
by (*simp add*: *size1_size*)

lemma *finite_set_tree*[*simp*]: *finite*(*set_tree t*)
by(*induction t*) *auto*

end

11 Function *isin* for Tree2

theory *Isin2*
imports
Tree2
Cmp
Set_Specs
begin

fun *isin* :: ('*a*::*linorder*,'*b*) *tree* ⇒ '*a* ⇒ *bool* **where**
isin Leaf x = *False* |
isin (*Node l a* _ *r*) *x* =
(*case cmp x a of*
LT ⇒ *isin l x* |
EQ ⇒ *True* |
GT ⇒ *isin r x*)

lemma *isin_set_inorder*: *sorted*(*inorder t*) ⟹ *isin t x* = (*x* ∈ *set*(*inorder t*))
by (*induction t*) (*auto simp*: *isin_simps*)

lemma *isin_set_tree*: *bst t* ⟹ *isin t x* ⟷ *x* ∈ *set_tree t*
by(*induction t*) *auto*

end

12 AVL Tree Implementation of Sets

theory *AVL_Set*
imports
Cmp
Isin2
HOL−Number_Theory.Fib
begin

type_synonym '*a avl_tree* = ('*a*,*nat*) *tree*

definition *empty* :: '*a avl_tree* **where**
empty = *Leaf*

Invariant:

fun *avl* :: '*a avl_tree* ⇒ *bool* **where**
avl Leaf = *True* |
avl (*Node l a h r*) =

```
 ((height l = height r ∨ height l = height r + 1 ∨ height r = height l + 1)
∧
  h = max (height l) (height r) + 1 ∧ avl l ∧ avl r)

fun ht :: 'a avl_tree ⇒ nat where
ht Leaf = 0 |
ht (Node l a h r) = h

definition node :: 'a avl_tree ⇒ 'a ⇒ 'a avl_tree ⇒ 'a avl_tree where
node l a r = Node l a (max (ht l) (ht r) + 1) r

definition balL :: 'a avl_tree ⇒ 'a ⇒ 'a avl_tree ⇒ 'a avl_tree where
balL l a r =
  (if ht l = ht r + 2 then
     case l of
       Node bl b _ br ⇒
         if ht bl < ht br then
           case br of
             Node cl c _ cr ⇒ node (node bl b cl) c (node cr a r)
         else node bl b (node br a r)
   else node l a r)

definition balR :: 'a avl_tree ⇒ 'a ⇒ 'a avl_tree ⇒ 'a avl_tree where
balR l a r =
   (if ht r = ht l + 2 then
      case r of
        Node bl b _ br ⇒
          if ht bl > ht br then
            case bl of
              Node cl c _ cr ⇒ node (node l a cl) c (node cr b br)
          else node (node l a bl) b br
  else node l a r)

fun insert :: 'a::linorder ⇒ 'a avl_tree ⇒ 'a avl_tree where
insert x Leaf = Node Leaf x 1 Leaf |
insert x (Node l a h r) = (case cmp x a of
   EQ ⇒ Node l a h r |
   LT ⇒ balL (insert x l) a r |
   GT ⇒ balR l a (insert x r))

fun split_max :: 'a avl_tree ⇒ 'a avl_tree * 'a where
split_max (Node l a _ r) =
  (if r = Leaf then (l,a) else let (r',a') = split_max r in (balL l a r', a'))
```

lemmas *split_max_induct* = *split_max.induct*[*case_names Node Leaf*]

fun *del_root* :: *'a avl_tree* ⇒ *'a avl_tree* **where**
del_root (*Node Leaf a h r*) = *r* |
del_root (*Node l a h Leaf*) = *l* |
del_root (*Node l a h r*) = (*let* (*l'*, *a'*) = *split_max l in balR l' a' r*)

lemmas *del_root_cases* = *del_root.cases*[*case_names Leaf_t Node_Leaf Node_Node*]

fun *delete* :: *'a*::*linorder* ⇒ *'a avl_tree* ⇒ *'a avl_tree* **where**
delete _ *Leaf* = *Leaf* |
delete x (*Node l a h r*) =
(*case cmp x a of*
EQ ⇒ *del_root* (*Node l a h r*) |
LT ⇒ *balR* (*delete x l*) *a r* |
GT ⇒ *balL l a* (*delete x r*))

12.1 Functional Correctness Proofs

Very different from the AFP/AVL proofs

12.1.1 Proofs for insert

lemma *inorder_balL*:
inorder (*balL l a r*) = *inorder l* @ *a* # *inorder r*
by (*auto simp*: *node_def balL_def split*:*tree.splits*)

lemma *inorder_balR*:
inorder (*balR l a r*) = *inorder l* @ *a* # *inorder r*
by (*auto simp*: *node_def balR_def split*:*tree.splits*)

theorem *inorder_insert*:
sorted(*inorder t*) ⟹ *inorder*(*insert x t*) = *ins_list x* (*inorder t*)
by (*induct t*)
(*auto simp*: *ins_list_simps inorder_balL inorder_balR*)

12.1.2 Proofs for delete

lemma *inorder_split_maxD*:
⟦ *split_max t* = (*t'*,*a*); *t* ≠ *Leaf* ⟧ ⟹
inorder t' @ [*a*] = *inorder t*
by(*induction t arbitrary*: *t' rule*: *split_max.induct*)
(*auto simp*: *inorder_balL split*: *if_splits prod.splits tree.split*)

lemma *inorder_del_root*:
inorder (*del_root* (*Node l a h r*)) = *inorder l* @ *inorder r*
by(*cases Node l a h r rule*: *del_root.cases*)
(*auto simp*: *inorder_balL inorder_balR inorder_split_maxD split*: *if_splits prod.splits*)

theorem *inorder_delete*:
sorted(*inorder t*) ⟹ *inorder* (*delete x t*) = *del_list x* (*inorder t*)
by(*induction t*)
(*auto simp*: *del_list_simps inorder_balL inorder_balR*
inorder_del_root inorder_split_maxD split: *prod.splits*)

12.2 AVL invariants

Essentially the AFP/AVL proofs

12.2.1 Insertion maintains AVL balance

declare *Let_def* [*simp*]

lemma [*simp*]: *avl t* ⟹ *ht t* = *height t*
by (*induct t*) *simp_all*

lemma *height_balL*:
⟦ *height l* = *height r* + *2*; *avl l*; *avl r* ⟧ ⟹
height (*balL l a r*) = *height r* + *2* ∨
height (*balL l a r*) = *height r* + *3*
by (*cases l*) (*auto simp*:*node_def balL_def split*:*tree.split*)

lemma *height_balR*:
⟦ *height r* = *height l* + *2*; *avl l*; *avl r* ⟧ ⟹
height (*balR l a r*) = *height l* + *2* ∨
height (*balR l a r*) = *height l* + *3*
by (*cases r*) (*auto simp add*:*node_def balR_def split*:*tree.split*)

lemma [*simp*]: *height*(*node l a r*) = *max* (*height l*) (*height r*) + *1*
by (*simp add*: *node_def*)

lemma *avl_node*:
⟦ *avl l*; *avl r*;
height l = *height r* ∨ *height l* = *height r* + *1* ∨ *height r* = *height l* + *1*
⟧ ⟹ *avl*(*node l a r*)
by (*auto simp add*:*max_def node_def*)

lemma *height_balL2*:
 ⟦ *avl l*; *avl r*; *height l* ≠ *height r* + *2* ⟧ ⟹
  height (*balL l a r*) = (*1* + *max* (*height l*) (*height r*))
by (*cases l*, *cases r*) (*simp_all add*: *balL_def*)

lemma *height_balR2*:
 ⟦ *avl l*; *avl r*; *height r* ≠ *height l* + *2* ⟧ ⟹
  height (*balR l a r*) = (*1* + *max* (*height l*) (*height r*))
by (*cases l*, *cases r*) (*simp_all add*: *balR_def*)

lemma *avl_balL*:
 assumes *avl l avl r* **and** *height l* = *height r* ∨ *height l* = *height r* + *1*
  ∨ *height r* = *height l* + *1* ∨ *height l* = *height r* + *2*
 shows *avl*(*balL l a r*)
proof(*cases l*)
 case *Leaf*
 with *assms* **show** *?thesis* **by** (*simp add*: *node_def balL_def*)
next
 case *Node*
 with *assms* **show** *?thesis*
 proof(*cases height l* = *height r* + *2*)
  case *True*
  from *True Node assms* **show** *?thesis*
   by (*auto simp*: *balL_def intro*!: *avl_node split*: *tree.split*) *arith*+
 next
  case *False*
  with *assms* **show** *?thesis* **by** (*simp add*: *avl_node balL_def*)
 qed
qed

lemma *avl_balR*:
 assumes *avl l* **and** *avl r* **and** *height l* = *height r* ∨ *height l* = *height r* +
1
  ∨ *height r* = *height l* + *1* ∨ *height r* = *height l* + *2*
 shows *avl*(*balR l a r*)
proof(*cases r*)
 case *Leaf*
 with *assms* **show** *?thesis* **by** (*simp add*: *node_def balR_def*)
next
 case *Node*
 with *assms* **show** *?thesis*
 proof(*cases height r* = *height l* + *2*)
  case *True*
   from *True Node assms* **show** *?thesis*

```
      by (auto simp: balR_def intro!: avl_node split: tree.split) arith+
  next
    case False
    with assms show ?thesis by (simp add: balR_def avl_node)
  qed
qed
```

Insertion maintains the AVL property:

```
theorem avl_insert:
  assumes avl t
  shows avl(insert x t)
        (height (insert x t) = height t ∨ height (insert x t) = height t + 1)
using assms
proof (induction t)
  case (Node l a h r)
  case 1
  show ?case
  proof(cases x = a)
    case True with Node 1 show ?thesis by simp
  next
    case False
    show ?thesis
    proof(cases x<a)
      case True with Node 1 show ?thesis by (auto simp add:avl_balL)
    next
    case False with Node 1 ‹x≠a› show ?thesis by (auto simp add:avl_balR)
    qed
  qed
  case 2
  show ?case
  proof(cases x = a)
    case True with Node 1 show ?thesis by simp
  next
    case False
    show ?thesis
    proof(cases x<a)
      case True
      show ?thesis
      proof(cases height (insert x l) = height r + 2)
          case False with Node 2 ‹x < a› show ?thesis by (auto simp:
height_balL2)
      next
        case True
        hence (height (balL (insert x l) a r) = height r + 2) ∨
```

```
        (height (balL (insert x l) a r) = height r + 3) (is ?A ∨ ?B)
        using Node 2 by (intro height_balL) simp_all
      thus ?thesis
      proof
        assume ?A with 2 ⟨x < a⟩ show ?thesis by (auto)
      next
       assume ?B with True 1 Node(2) ⟨x < a⟩ show ?thesis by (simp)
arith
      qed
    qed
  next
    case False
    show ?thesis
    proof(cases height (insert x r) = height l + 2)
       case False with Node 2 ⟨¬x < a⟩ show ?thesis by (auto simp:
height_balR2)
    next
      case True
      hence (height (balR l a (insert x r)) = height l + 2) ∨
        (height (balR l a (insert x r)) = height l + 3)  (is ?A ∨ ?B)
        using Node 2 by (intro height_balR) simp_all
      thus ?thesis
      proof
        assume ?A with 2 ⟨¬x < a⟩ show ?thesis by (auto)
      next
       assume ?B with True 1 Node(4) ⟨¬x < a⟩ show ?thesis by (simp)
arith
      qed
    qed
  qed
qed
qed simp_all
```

12.2.2 Deletion maintains AVL balance

```
lemma avl_split_max:
  assumes avl x and x ≠ Leaf
  shows avl (fst (split_max x)) height x = height(fst (split_max x)) ∨
        height x = height(fst (split_max x)) + 1
using assms
proof (induct x rule: split_max_induct)
  case (Node l a h r)
  case 1
  thus ?case using Node
```

```
    by (auto simp: height_balL height_balL2 avl_balL split:prod.split)
next
  case (Node l a h r)
  case 2
  let ?r' = fst (split_max r)
  from ⟨avl x⟩ Node 2 have avl l and avl r by simp_all
  thus ?case using Node 2 height_balL[of l ?r' a] height_balL2[of l ?r' a]
    apply (auto split:prod.splits simp del:avl.simps) by arith+
qed auto

lemma avl_del_root:
  assumes avl t and t ≠ Leaf
  shows avl(del_root t)
using assms
proof (cases t rule:del_root_cases)
  case (Node_Node ll ln lh lr n h rl rn rh rr)
  let ?l = Node ll ln lh lr
  let ?r = Node rl rn rh rr
  let ?l' = fst (split_max ?l)
  from ⟨avl t⟩ and Node_Node have avl ?r by simp
  from ⟨avl t⟩ and Node_Node have avl ?l by simp
  hence avl(?l') height ?l = height(?l') ∨
        height ?l = height(?l') + 1 by (rule avl_split_max,simp)+
  with ⟨avl t⟩ Node_Node have height ?l' = height ?r ∨ height ?l' = height
?r + 1
            ∨ height ?r = height ?l' + 1 ∨ height ?r = height ?l' + 2 by
fastforce
  with ⟨avl ?l'⟩ ⟨avl ?r⟩ have avl(balR ?l' (snd(split_max ?l)) ?r)
    by (rule avl_balR)
  with Node_Node show ?thesis by (auto split:prod.splits)
qed simp_all

lemma height_del_root:
  assumes avl t and t ≠ Leaf
  shows height t = height(del_root t) ∨ height t = height(del_root t) + 1
using assms
proof (cases t rule: del_root_cases)
  case (Node_Node ll ln lh lr n h rl rn rh rr)
  let ?l = Node ll ln lh lr
  let ?r = Node rl rn rh rr
  let ?l' = fst (split_max ?l)
  let ?t' = balR ?l' (snd(split_max ?l)) ?r
  from ⟨avl t⟩ and Node_Node have avl ?r by simp
  from ⟨avl t⟩ and Node_Node have avl ?l by simp
```

```
  hence avl(?l')  by (rule avl_split_max,simp)
  have l'_height: height ?l = height ?l' ∨ height ?l = height ?l' + 1 using
⟨avl ?l⟩ by (intro avl_split_max) auto
  have t_height: height t = 1 + max (height ?l) (height ?r) using ⟨avl t⟩
Node_Node by simp
  have height t = height ?t' ∨ height t = height ?t' + 1 using  ⟨avl t⟩
Node_Node
  proof(cases height ?r = height ?l' + 2)
    case False
    show ?thesis using l'_height t_height False
      by (subst height_balR2[OF ⟨avl ?l'⟩ ⟨avl ?r⟩ False])+ arith
  next
    case True
    show ?thesis
   proof(cases rule: disjE[OF height_balR[OF True ⟨avl ?l'⟩ ⟨avl ?r⟩, of snd
(split_max ?l)]])
      case 1 thus ?thesis using l'_height t_height True by arith
    next
      case 2 thus ?thesis using l'_height t_height True by arith
    qed
  qed
  thus ?thesis using Node_Node by (auto split:prod.splits)
qed simp_all
```

Deletion maintains the AVL property:

```
theorem avl_delete:
  assumes avl t
  shows avl(delete x t) and height t = (height (delete x t)) ∨ height t =
height (delete x t) + 1
using assms
proof (induct t)
  case (Node l n h r)
  case 1
  show ?case
  proof(cases x = n)
    case True with Node 1 show ?thesis by (auto simp:avl_del_root)
  next
    case False
    show ?thesis
    proof(cases x<n)
      case True with Node 1 show ?thesis by (auto simp add:avl_balR)
    next
    case False with Node 1 ⟨x≠n⟩ show ?thesis by (auto simp add:avl_balL)
    qed
```

```
  qed
  case 2
  show ?case
  proof(cases x = n)
    case True
    with 1 have height (Node l n h r) = height(del_root (Node l n h r))
      ∨ height (Node l n h r) = height(del_root (Node l n h r)) + 1
      by (subst height_del_root,simp_all)
    with True show ?thesis by simp
  next
    case False
    show ?thesis
    proof(cases x<n)
      case True
      show ?thesis
      proof(cases height r = height (delete x l) + 2)
      case False with Node 1 ‹x < n› show ?thesis by(auto simp: balR_def)
      next
        case True
        hence (height (balR (delete x l) n r) = height (delete x l) + 2) ∨
          height (balR (delete x l) n r) = height (delete x l) + 3 (is ?A ∨
?B)
          using Node 2 by (intro height_balR) auto
        thus ?thesis
        proof
           assume ?A with ‹x < n› Node 2 show ?thesis by(auto simp:
balR_def)
        next
           assume ?B with ‹x < n› Node 2 show ?thesis by(auto simp:
balR_def)
        qed
      qed
    next
      case False
      show ?thesis
      proof(cases height l = height (delete x r) + 2)
        case False with Node 1 ‹¬x < n› ‹x ≠ n› show ?thesis by(auto
simp: balL_def)
      next
        case True
        hence (height (balL l n (delete x r)) = height (delete x r) + 2) ∨
          height (balL l n (delete x r)) = height (delete x r) + 3 (is ?A ∨
?B)
          using Node 2 by (intro height_balL) auto
```

thus *?thesis*
proof
assume *?A* **with** ‹$\neg x < n$› ‹$x \neq n$› *Node 2* **show** *?thesis* **by**(*auto simp*: *balL_def*)
next
assume *?B* **with** ‹$\neg x < n$› ‹$x \neq n$› *Node 2* **show** *?thesis* **by**(*auto simp*: *balL_def*)
qed
qed
qed
qed
qed *simp_all*

12.3 Overall correctness

interpretation *S*: *Set_by_Ordered*
where *empty* = *empty* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = *avl*
proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** (*simp add*: *empty_def*)
next
case *2* **thus** *?case* **by**(*simp add*: *isin_set_inorder*)
next
case *3* **thus** *?case* **by**(*simp add*: *inorder_insert*)
next
case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
case *5* **thus** *?case* **by** (*simp add*: *empty_def*)
next
case *6* **thus** *?case* **by** (*simp add*: *avl_insert*(*1*))
next
case *7* **thus** *?case* **by** (*simp add*: *avl_delete*(*1*))
qed

12.4 Height-Size Relation

Based on theorems by Daniel Stüwe, Manuel Eberl and Peter Lammich.

lemma *height_invers*:
(*height t* = *0*) = (*t* = *Leaf*)
avl t ⟹ (*height t* = *Suc h*) = (∃ *l a r* . *t* = *Node l a* (*Suc h*) *r*)
by (*induction t*) *auto*

Any AVL tree of height h has at least *fib* (h+*2*) leaves:

lemma *avl_fib_bound*: *avl t* $\Longrightarrow$ *height t* = *h* $\Longrightarrow$ *fib* (*h*+*2*) $\leq$ *size1 t*
proof (*induction h arbitrary*: *t rule*: *fib.induct*)
case *1* **thus** *?case* **by** (*simp add*: *height_invers*)
next
case *2* **thus** *?case* **by** (*cases t*) (*auto simp*: *height_invers*)
next
case (*3 h*)
from *3.prems* **obtain** *l a r* **where**
[*simp*]: *t* = *Node l a* (*Suc*(*Suc h*)) *r avl l avl r*
and *C*:
height r = *Suc h* $\wedge$ *height l* = *Suc h*
$\vee$ *height r* = *Suc h* $\wedge$ *height l* = *h*
$\vee$ *height r* = *h* $\wedge$ *height l* = *Suc h* (**is** *?C1* $\vee$ *?C2* $\vee$ *?C3*)
by (*cases t*) (*simp*, *fastforce*)
{
assume *?C1*
with *3.IH*(*1*)
have *fib* (*h* + *3*) $\leq$ *size1 l fib* (*h* + *3*) $\leq$ *size1 r*
by (*simp_all add*: *eval_nat_numeral*)
hence *?case* **by** (*auto simp*: *eval_nat_numeral*)
} moreover {
assume *?C2*
hence *?case* **using** *3.IH*(*1*)[*of r*] *3.IH*(*2*)[*of l*] **by** *auto*
} moreover {
assume *?C3*
hence *?case* **using** *3.IH*(*1*)[*of l*] *3.IH*(*2*)[*of r*] **by** *auto*
} ultimately show *?case* **using** *C* **by** *blast*
qed

lemma *fib_alt_induct* [*consumes 1*, *case_names 1 2 rec*]:
assumes *n* > *0 P 1 P 2* $\bigwedge$*n*. *n* > *0* $\Longrightarrow$ *P n* $\Longrightarrow$ *P* (*Suc n*) $\Longrightarrow$ *P* (*Suc* (*Suc n*))
shows *P n*
using *assms*(*1*)
proof (*induction n rule*: *fib.induct*)
case (*3 n*)
thus *?case* **using** *assms* **by** (*cases n*) (*auto simp*: *eval_nat_numeral*)
qed (*insert assms*, *auto*)

An exponential lower bound for *fib*:

lemma *fib_lowerbound*:
defines φ $\equiv$ (*1* + *sqrt 5*) / *2*
defines *c* $\equiv$ *1* / φ ^ *2*
assumes *n* > *0*

shows *real (fib n) ≥ c * φ ^ n*
proof −
have *φ > 1* **by** (*simp add*: *φ_def*)
hence *c > 0* **by** (*simp add*: *c_def*)
from ‹*n > 0*› **show** *?thesis*
proof (*induction n rule*: *fib_alt_induct*)
case (*rec n*)
have *c * φ ^ Suc (Suc n) = φ ^ 2 * (c * φ ^ n)*
by (*simp add*: *field_simps power2_eq_square*)
also have *... ≤ (φ + 1) * (c * φ ^ n)*
by (*rule mult_right_mono*) (*insert* ‹*c > 0*›, *simp_all add*: *φ_def power2_eq_square field_simps*)
also have *... = c * φ ^ Suc n + c * φ ^ n*
by (*simp add*: *field_simps*)
also have *... ≤ real (fib (Suc n)) + real (fib n)*
by (*intro add_mono rec.IH*)
finally show *?case* **by** *simp*
qed (*insert* ‹*φ > 1*›, *simp_all add*: *c_def power2_eq_square eval_nat_numeral*)
qed

The size of an AVL tree is (at least) exponential in its height:

lemma *avl_size_lowerbound*:
defines *φ ≡ (1 + sqrt 5) / 2*
assumes *avl t*
shows *φ ^ (height t) ≤ size1 t*
proof −
have *φ > 0* **by**(*simp add*: *φ_def add_pos_nonneg*)
hence *φ ^ height t = (1 / φ ^ 2) * φ ^ (height t + 2)*
by(*simp add*: *field_simps power2_eq_square*)
also have *... ≤ fib (height t + 2)*
using *fib_lowerbound*[*of height t + 2*] **by**(*simp add*: *φ_def*)
also have *... ≤ size1 t*
using *avl_fib_bound*[*of t height t*] *assms* **by** *simp*
finally show *?thesis* **.**
qed

The height of an AVL tree is most *1 / log 2 φ ≈ 1.44* times worse than *log 2 (real (size1 t))*:

lemma *avl_height_upperbound*:
defines *φ ≡ (1 + sqrt 5) / 2*
assumes *avl t*
shows *height t ≤ (1/log 2 φ) * log 2 (size1 t)*
proof −
have *φ > 0 φ > 1* **by**(*auto simp*: *φ_def pos_add_strict*)

hence *height t = log φ (φ ^ height t)* **by**(*simp add*: *log_nat_power*)
also have ... ≤ *log φ* (*size1 t*)
using *avl_size_lowerbound*[*OF assms*(*2*), *folded φ_def*] ‹*1 < φ*› **by** *simp*
also have ... = (*1*/*log 2 φ*) * *log 2* (*size1 t*)
by(*simp add*: *log_base_change*[*of 2 φ*])
finally show *?thesis* **.**
qed

end

13 Function *lookup* for Tree2

theory *Lookup2*
imports
Tree2
Cmp
Map_Specs
begin

fun *lookup* :: ('*a*::*linorder* * '*b*, '*c*) *tree* ⇒ '*a* ⇒ '*b option* **where**
lookup Leaf x = None |
lookup (*Node l* (*a*,*b*) _ *r*) *x* =
(*case cmp x a of LT* ⇒ *lookup l x* | *GT* ⇒ *lookup r x* | *EQ* ⇒ *Some b*)

lemma *lookup_map_of*:
sorted1(*inorder t*) ⟹ *lookup t x = map_of* (*inorder t*) *x*
by(*induction t*) (*auto simp*: *map_of_simps split*: *option.split*)

end

14 AVL Tree Implementation of Maps

theory *AVL_Map*
imports
AVL_Set
Lookup2
begin

fun *update* :: '*a*::*linorder* ⇒ '*b* ⇒ ('*a*∗'*b*) *avl_tree* ⇒ ('*a*∗'*b*) *avl_tree* **where**
update x y Leaf = Node Leaf (*x*,*y*) *1 Leaf* |
update x y (*Node l* (*a*,*b*) *h r*) = (*case cmp x a of*
EQ ⇒ *Node l* (*x*,*y*) *h r* |
LT ⇒ *balL* (*update x y l*) (*a*,*b*) *r* |

```
  GT ⇒ balR l (a,b) (update x y r))

fun delete :: 'a::linorder ⇒ ('a*'b) avl_tree ⇒ ('a*'b) avl_tree where
delete _ Leaf = Leaf |
delete x (Node l (a,b) h r) = (case cmp x a of
  EQ ⇒ del_root (Node l (a,b) h r) |
  LT ⇒ balR (delete x l) (a,b) r |
  GT ⇒ balL l (a,b) (delete x r))
```

14.1 Functional Correctness

```
theorem inorder_update:
  sorted1(inorder t) ⟹ inorder(update x y t) = upd_list x y (inorder t)
by (induct t) (auto simp: upd_list_simps inorder_balL inorder_balR)


theorem inorder_delete:
  sorted1(inorder t) ⟹ inorder (delete x t) = del_list x (inorder t)
by(induction t)
  (auto simp: del_list_simps inorder_balL inorder_balR
     inorder_del_root inorder_split_maxD split: prod.splits)
```

14.2 AVL invariants

14.2.1 Insertion maintains AVL balance

```
theorem avl_update:
  assumes avl t
  shows avl(update x y t)
        (height (update x y t) = height t ∨ height (update x y t) = height t
+ 1)
using assms
proof (induction x y t rule: update.induct)
  case eq2: (2 x y l a b h r)
  case 1
  show ?case
  proof(cases x = a)
    case True with eq2 1 show ?thesis by simp
  next
    case False
    with eq2 1 show ?thesis
    proof(cases x<a)
      case True with eq2 1 show ?thesis by (auto simp add:avl_balL)
    next
     case False with eq2 1 ‹x≠a› show ?thesis by (auto simp add:avl_balR)
```

```
    qed
  qed
  case 2
  show ?case
  proof(cases x = a)
    case True with eq2 1 show ?thesis by simp
  next
    case False
    show ?thesis
    proof(cases x<a)
      case True
      show ?thesis
      proof(cases height (update x y l) = height r + 2)
          case False with eq2 2 ‹x < a› show ?thesis by (auto simp:
height_balL2)
      next
        case True
        hence (height (balL (update x y l) (a,b) r) = height r + 2) ∨
          (height (balL (update x y l) (a,b) r) = height r + 3) (is ?A ∨ ?B)
          using eq2 2 ‹x<a› by (intro height_balL) simp_all
        thus ?thesis
        proof
          assume ?A with 2 ‹x < a› show ?thesis by (auto)
        next
          assume ?B with True 1 eq2(2) ‹x < a› show ?thesis by (simp)
arith
        qed
      qed
    next
      case False
      show ?thesis
      proof(cases height (update x y r) = height l + 2)
          case False with eq2 2 ‹¬x < a› show ?thesis by (auto simp:
height_balR2)
      next
        case True
        hence (height (balR l (a,b) (update x y r)) = height l + 2) ∨
          (height (balR l (a,b) (update x y r)) = height l + 3)  (is ?A ∨ ?B)
          using eq2 2 ‹¬x < a› ‹x ≠ a› by (intro height_balR) simp_all
        thus ?thesis
        proof
          assume ?A with 2 ‹¬x < a› show ?thesis by (auto)
        next
          assume ?B with True 1 eq2(4) ‹¬x < a› show ?thesis by (simp)
```

arith
qed
qed
qed
qed
qed *simp_all*

14.2.2 Deletion maintains AVL balance

theorem *avl_delete*:
assumes *avl t*
shows *avl*(*delete x t*) **and** *height t* = (*height* (*delete x t*)) ∨ *height t* = *height* (*delete x t*) + *1*
using *assms*
proof (*induct t*)
case (*Node l n h r*)
obtain *a b* **where** [*simp*]: *n* = (*a*,*b*) **by** *fastforce*
case *1*
show *?case*
proof(*cases x* = *a*)
case *True* **with** *Node 1* **show** *?thesis* **by** (*auto simp*:*avl_del_root*)
next
case *False*
show *?thesis*
proof(*cases x*<*a*)
case *True* **with** *Node 1* **show** *?thesis* **by** (*auto simp add*:*avl_balR*)
next
case *False* **with** *Node 1* ‹*x*≠*a*› **show** *?thesis* **by** (*auto simp add*:*avl_balL*)
qed
qed
case *2*
show *?case*
proof(*cases x* = *a*)
case *True*
with *1* **have** *height* (*Node l n h r*) = *height*(*del_root* (*Node l n h r*))
∨ *height* (*Node l n h r*) = *height*(*del_root* (*Node l n h r*)) + *1*
by (*subst height_del_root*,*simp_all*)
with *True* **show** *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof(*cases x*<*a*)
case *True*
show *?thesis*

```
    proof(cases height r = height (delete x l) + 2)
    case False with Node 1 ‹x < a› show ?thesis by(auto simp: balR_def)
    next
      case True
      hence (height (balR (delete x l) n r) = height (delete x l) + 2) ∨
        height (balR (delete x l) n r) = height (delete x l) + 3 (is ?A ∨
?B)
        using Node 2 by (intro height_balR) auto
      thus ?thesis
      proof
        assume ?A with ‹x < a› Node 2 show ?thesis by(auto simp:
balR_def)
      next
        assume ?B with ‹x < a› Node 2 show ?thesis by(auto simp:
balR_def)
      qed
    qed
  next
    case False
    show ?thesis
    proof(cases height l = height (delete x r) + 2)
      case False with Node 1 ‹¬x < a› ‹x ≠ a› show ?thesis by(auto
simp: balL_def)
    next
      case True
      hence (height (balL l n (delete x r)) = height (delete x r) + 2) ∨
        height (balL l n (delete x r)) = height (delete x r) + 3 (is ?A ∨
?B)
        using Node 2 by (intro height_balL) auto
      thus ?thesis
      proof
        assume ?A with ‹¬x < a› ‹x ≠ a› Node 2 show ?thesis by(auto
simp: balL_def)
      next
        assume ?B with ‹¬x < a› ‹x ≠ a› Node 2 show ?thesis by(auto
simp: balL_def)
      qed
    qed
  qed
qed simp_all


interpretation M: Map_by_Ordered
```

where *empty* = *empty* **and** *lookup* = *lookup* **and** *update* = *update* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = *avl*
proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** (*simp add*: *empty_def*)
next
case *2* **thus** *?case* **by**(*simp add*: *lookup_map_of*)
next
case *3* **thus** *?case* **by**(*simp add*: *inorder_update*)
next
case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
case *5* **show** *?case* **by** (*simp add*: *empty_def*)
next
case *6* **thus** *?case* **by**(*simp add*: *avl_update*(*1*))
next
case *7* **thus** *?case* **by**(*simp add*: *avl_delete*(*1*))
qed

end

15 Red-Black Trees

theory *RBT*
imports *Tree2*
begin

datatype *color* = *Red* | *Black*

type_synonym *'a rbt* = (*'a*,*color*)*tree*

abbreviation *R* **where** *R l a r* ≡ *Node l a Red r*
abbreviation *B* **where** *B l a r* ≡ *Node l a Black r*

fun *baliL* :: *'a rbt* ⇒ *'a* ⇒ *'a rbt* ⇒ *'a rbt* **where**
baliL (*R* (*R t1 a1 t2*) *a2 t3*) *a3 t4* = *R* (*B t1 a1 t2*) *a2* (*B t3 a3 t4*) |
baliL (*R t1 a1* (*R t2 a2 t3*)) *a3 t4* = *R* (*B t1 a1 t2*) *a2* (*B t3 a3 t4*) |
baliL t1 a t2 = *B t1 a t2*

fun *baliR* :: *'a rbt* ⇒ *'a* ⇒ *'a rbt* ⇒ *'a rbt* **where**
baliR t1 a1 (*R t2 a2* (*R t3 a3 t4*)) = *R* (*B t1 a1 t2*) *a2* (*B t3 a3 t4*) |
baliR t1 a1 (*R* (*R t2 a2 t3*) *a3 t4*) = *R* (*B t1 a1 t2*) *a2* (*B t3 a3 t4*) |
baliR t1 a t2 = *B t1 a t2*

fun *paint* :: *color* ⇒ *'a rbt* ⇒ *'a rbt* **where**
paint c Leaf = *Leaf* |
paint c (*Node l a _ r*) = *Node l a c r*

fun *baldL* :: *'a rbt* ⇒ *'a* ⇒ *'a rbt* ⇒ *'a rbt* **where**
baldL (*R t1 x t2*) *y t3* = *R* (*B t1 x t2*) *y t3* |
baldL bl x (*B t1 y t2*) = *baliR bl x* (*R t1 y t2*) |
baldL bl x (*R* (*B t1 y t2*) *z t3*) = *R* (*B bl x t1*) *y* (*baliR t2 z* (*paint Red t3*)) |
baldL t1 x t2 = *R t1 x t2*

fun *baldR* :: *'a rbt* ⇒ *'a* ⇒ *'a rbt* ⇒ *'a rbt* **where**
baldR t1 x (*R t2 y t3*) = *R t1 x* (*B t2 y t3*) |
baldR (*B t1 x t2*) *y t3* = *baliL* (*R t1 x t2*) *y t3* |
baldR (*R t1 x* (*B t2 y t3*)) *z t4* = *R* (*baliL* (*paint Red t1*) *x t2*) *y* (*B t3 z t4*) |
baldR t1 x t2 = *R t1 x t2*

fun *combine* :: *'a rbt* ⇒ *'a rbt* ⇒ *'a rbt* **where**
combine Leaf t = *t* |
combine t Leaf = *t* |
combine (*R t1 a t2*) (*R t3 c t4*) =
  (*case combine 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*)) |
combine (*B t1 a t2*) (*B t3 c t4*) =
  (*case combine t2 t3 of*
    R t2' b t3' ⇒ *R* (*B t1 a t2'*) *b* (*B t3' c t4*) |
    t23 ⇒ *baldL t1 a* (*B t23 c t4*)) |
combine t1 (*R t2 a t3*) = *R* (*combine t1 t2*) *a t3* |
combine (*R t1 a t2*) *t3* = *R t1 a* (*combine t2 t3*)

end

16 Red-Black Tree Implementation of Sets

theory *RBT_Set*
imports
  Complex_Main
  RBT
  Cmp
  Isin2

begin

definition *empty* :: *'a rbt* **where**
empty = *Leaf*

fun *ins* :: *'a*::*linorder* ⇒ *'a rbt* ⇒ *'a rbt* **where**
ins x Leaf = *R Leaf x Leaf* |
ins x (*B l a r*) =
(*case cmp x a of*
LT ⇒ *baliL* (*ins x l*) *a r* |
GT ⇒ *baliR l a* (*ins x r*) |
EQ ⇒ *B l a r*) |
ins x (*R l a r*) =
(*case cmp x a of*
LT ⇒ *R* (*ins x l*) *a r* |
GT ⇒ *R l a* (*ins x r*) |
EQ ⇒ *R l a r*)

definition *insert* :: *'a*::*linorder* ⇒ *'a rbt* ⇒ *'a rbt* **where**
insert x t = *paint Black* (*ins x t*)

fun *color* :: *'a rbt* ⇒ *color* **where**
color Leaf = *Black* |
color (*Node* _ _ *c* _) = *c*

fun *del* :: *'a*::*linorder* ⇒ *'a rbt* ⇒ *'a rbt* **where**
del x Leaf = *Leaf* |
del x (*Node l a* _ *r*) =
(*case cmp x a of*
LT ⇒ *if l* ≠ *Leaf* ∧ *color l* = *Black*
then baldL (*del x l*) *a r else R* (*del x l*) *a r* |
GT ⇒ *if r* ≠ *Leaf*∧ *color r* = *Black*
then baldR l a (*del x r*) *else R l a* (*del x r*) |
EQ ⇒ *combine l r*)

definition *delete* :: *'a*::*linorder* ⇒ *'a rbt* ⇒ *'a rbt* **where**
delete x t = *paint Black* (*del x t*)

16.1 Functional Correctness Proofs

lemma *inorder_paint*: *inorder*(*paint c t*) = *inorder t*
by(*cases t*) (*auto*)

lemma *inorder_baliL*:

inorder(*baliL l a r*) = *inorder l* @ *a* # *inorder r*
by(*cases* (*l*,*a*,*r*) *rule*: *baliL.cases*) (*auto*)

lemma *inorder_baliR*:
inorder(*baliR l a r*) = *inorder l* @ *a* # *inorder r*
by(*cases* (*l*,*a*,*r*) *rule*: *baliR.cases*) (*auto*)

lemma *inorder_ins*:
sorted(*inorder t*) ⟹ *inorder*(*ins x t*) = *ins_list x* (*inorder t*)
by(*induction x t rule*: *ins.induct*)
(*auto simp*: *ins_list_simps inorder_baliL inorder_baliR*)

lemma *inorder_insert*:
sorted(*inorder t*) ⟹ *inorder*(*insert x t*) = *ins_list x* (*inorder t*)
by (*simp add*: *insert_def inorder_ins inorder_paint*)

lemma *inorder_baldL*:
inorder(*baldL l a r*) = *inorder l* @ *a* # *inorder r*
by(*cases* (*l*,*a*,*r*) *rule*: *baldL.cases*)
(*auto simp*: *inorder_baliL inorder_baliR inorder_paint*)

lemma *inorder_baldR*:
inorder(*baldR l a r*) = *inorder l* @ *a* # *inorder r*
by(*cases* (*l*,*a*,*r*) *rule*: *baldR.cases*)
(*auto simp*: *inorder_baliL inorder_baliR inorder_paint*)

lemma *inorder_combine*:
inorder(*combine l r*) = *inorder l* @ *inorder r*
by(*induction l r rule*: *combine.induct*)
(*auto simp*: *inorder_baldL inorder_baldR split*: *tree.split color.split*)

lemma *inorder_del*:
sorted(*inorder t*) ⟹ *inorder*(*del x t*) = *del_list x* (*inorder t*)
by(*induction x t rule*: *del.induct*)
(*auto simp*: *del_list_simps inorder_combine inorder_baldL inorder_baldR*)

lemma *inorder_delete*:
sorted(*inorder t*) ⟹ *inorder*(*delete x t*) = *del_list x* (*inorder t*)
by (*auto simp*: *delete_def inorder_del inorder_paint*)

16.2 Structural invariants

The proofs are due to Markus Reiter and Alexander Krauss.

fun *bheight* :: *'a rbt* ⇒ *nat* **where**

bheight Leaf = 0 |
bheight (Node l x c r) = (if c = Black then bheight l + 1 else bheight l)

fun *invc* :: *'a rbt ⇒ bool* **where**
invc Leaf = True |
invc (Node l a c r) =
(invc l ∧ invc r ∧ (c = Red ⟶ color l = Black ∧ color r = Black))

fun *invc2* :: *'a rbt ⇒ bool* — Weaker version **where**
invc2 Leaf = True |
invc2 (Node l a c r) = (invc l ∧ invc r)

fun *invh* :: *'a rbt ⇒ bool* **where**
invh Leaf = True |
invh (Node l x c r) = (invh l ∧ invh r ∧ bheight l = bheight r)

lemma *invc2I*: *invc t ⟹ invc2 t*
by *(cases t) simp+*

definition *rbt* :: *'a rbt ⇒ bool* **where**
rbt t = (invc t ∧ invh t ∧ color t = Black)

lemma *color_paint_Black*: *color (paint Black t) = Black*
by *(cases t) auto*

lemma *paint_invc2*: *invc2 t ⟹ invc2 (paint c t)*
by *(cases t) auto*

lemma *invc_paint_Black*: *invc2 t ⟹ invc (paint Black t)*
by *(cases t) auto*

lemma *invh_paint*: *invh t ⟹ invh (paint c t)*
by *(cases t) auto*

lemma *invc_baliL*:
⟦invc2 l; invc r⟧ ⟹ invc (baliL l a r)
by *(induct l a r rule: baliL.induct) auto*

lemma *invc_baliR*:
⟦invc l; invc2 r⟧ ⟹ invc (baliR l a r)
by *(induct l a r rule: baliR.induct) auto*

lemma *bheight_baliL*:
bheight l = bheight r ⟹ bheight (baliL l a r) = Suc (bheight l)

by (*induct l a r rule*: *baliL.induct*) *auto*

lemma *bheight_baliR*:
bheight l = *bheight r* ⟹ *bheight* (*baliR l a r*) = *Suc* (*bheight l*)
by (*induct l a r rule*: *baliR.induct*) *auto*

lemma *invh_baliL*:
⟦ *invh l*; *invh r*; *bheight l* = *bheight r* ⟧ ⟹ *invh* (*baliL l a r*)
by (*induct l a r rule*: *baliL.induct*) *auto*

lemma *invh_baliR*:
⟦ *invh l*; *invh r*; *bheight l* = *bheight r* ⟧ ⟹ *invh* (*baliR l a r*)
by (*induct l a r rule*: *baliR.induct*) *auto*

16.2.1 Insertion

lemma *invc_ins*: **assumes** *invc t*
shows *color t* = *Black* ⟹ *invc* (*ins x t*) *invc2* (*ins x t*)
using *assms*
by (*induct x t rule*: *ins.induct*) (*auto simp*: *invc_baliL invc_baliR invc2I*)

lemma *invh_ins*: **assumes** *invh t*
shows *invh* (*ins x t*) *bheight* (*ins x t*) = *bheight t*
using *assms*
by(*induct x t rule*: *ins.induct*)
(*auto simp*: *invh_baliL invh_baliR bheight_baliL bheight_baliR*)

theorem *rbt_insert*: *rbt t* ⟹ *rbt* (*insert x t*)
by (*simp add*: *invc_ins*(*2*) *invh_ins*(*1*) *color_paint_Black invc_paint_Black invh_paint*
rbt_def insert_def)

16.2.2 Deletion

lemma *bheight_paint_Red*:
color t = *Black* ⟹ *bheight* (*paint Red t*) = *bheight t* − *1*
by (*cases t*) *auto*

lemma *invh_baldL_invc*:
⟦ *invh l*; *invh r*; *bheight l* + *1* = *bheight r*; *invc r* ⟧
⟹ *invh* (*baldL l a r*) ∧ *bheight* (*baldL l a r*) = *bheight l* + *1*
by (*induct l a r rule*: *baldL.induct*)
(*auto simp*: *invh_baliR invh_paint bheight_baliR bheight_paint_Red*)

lemma *invh_baldL_Black*:
⟦ *invh l*; *invh r*; *bheight l + 1 = bheight r*; *color r = Black* ⟧
⟹ *invh* (*baldL l a r*) ∧ *bheight* (*baldL l a r*) = *bheight r*
by (*induct l a r rule*: *baldL.induct*) (*auto simp add*: *invh_baliR bheight_baliR*)

lemma *invc_baldL*: ⟦*invc2 l*; *invc r*; *color r = Black*⟧ ⟹ *invc* (*baldL l a r*)
by (*induct l a r rule*: *baldL.induct*) (*simp_all add*: *invc_baliR*)

lemma *invc2_baldL*: ⟦ *invc2 l*; *invc r* ⟧ ⟹ *invc2* (*baldL l a r*)
by (*induct l a r rule*: *baldL.induct*) (*auto simp*: *invc_baliR paint_invc2 invc2I*)

lemma *invh_baldR_invc*:
⟦ *invh l*; *invh r*; *bheight l = bheight r + 1*; *invc l* ⟧
⟹ *invh* (*baldR l a r*) ∧ *bheight* (*baldR l a r*) = *bheight l*
by(*induct l a r rule*: *baldR.induct*)
(*auto simp*: *invh_baliL bheight_baliL invh_paint bheight_paint_Red*)

lemma *invc_baldR*: ⟦*invc a*; *invc2 b*; *color a = Black*⟧ ⟹ *invc* (*baldR a x b*)
by (*induct a x b rule*: *baldR.induct*) (*simp_all add*: *invc_baliL*)

lemma *invc2_baldR*: ⟦ *invc l*; *invc2 r* ⟧ ⟹*invc2* (*baldR l x r*)
by (*induct l x r rule*: *baldR.induct*) (*auto simp*: *invc_baliL paint_invc2 invc2I*)

lemma *invh_combine*:
⟦ *invh l*; *invh r*; *bheight l = bheight r* ⟧
⟹ *invh* (*combine l r*) ∧ *bheight* (*combine l r*) = *bheight l*
by (*induct l r rule*: *combine.induct*)
(*auto simp*: *invh_baldL_Black split*: *tree.splits color.splits*)

lemma *invc_combine*:
assumes *invc l invc r*
shows *color l = Black* ⟹ *color r = Black* ⟹ *invc* (*combine l r*)
invc2 (*combine l r*)
using *assms*
by (*induct l r rule*: *combine.induct*)
(*auto simp*: *invc_baldL invc2I split*: *tree.splits color.splits*)

lemma *neq_LeafD*: *t* ≠ *Leaf* ⟹ ∃ *c l x r*. *t = Node c l x r*
by(*cases t*) *auto*

lemma *del_invc_invh*: *invh t* ⟹ *invc t* ⟹ *invh* (*del x t*) ∧

```
   (color t = Red ∧ bheight (del x t) = bheight t ∧ invc (del x t) ∨
    color t = Black ∧ bheight (del x t) = bheight t − 1 ∧ invc2 (del x t))
proof (induct x t rule: del.induct)
case (2 x _ y c)
  have x = y ∨ x < y ∨ x > y by auto
  thus ?case proof (elim disjE)
    assume x = y
    with 2 show ?thesis
    by (cases c) (simp_all add: invh_combine invc_combine)
  next
    assume x < y
    with 2 show ?thesis
      by(cases c)
        (auto simp: invh_baldL_invc invc_baldL invc2_baldL dest: neq_LeafD)
  next
    assume y < x
    with 2 show ?thesis
      by(cases c)
        (auto simp: invh_baldR_invc invc_baldR invc2_baldR dest: neq_LeafD)
  qed
qed auto

theorem rbt_delete: rbt t ⟹ rbt (delete k t)
by (metis delete_def rbt_def color_paint_Black del_invc_invh invc_paint_Black
invc2I invh_paint)
```

Overall correctness:

```
interpretation S: Set_by_Ordered
where empty = empty and isin = isin and insert = insert and delete =
delete
and inorder = inorder and inv = rbt
proof (standard, goal_cases)
  case 1 show ?case by (simp add: empty_def)
next
  case 2 thus ?case by(simp add: isin_set_inorder)
next
  case 3 thus ?case by(simp add: inorder_insert)
next
  case 4 thus ?case by(simp add: inorder_delete)
next
  case 5 thus ?case by (simp add: rbt_def empty_def)
next
  case 6 thus ?case by (simp add: rbt_insert)
next
```

case *7* **thus** *?case* **by** (*simp add*: *rbt_delete*)
qed

16.3 Height-Size Relation

lemma *neq_Black*[*simp*]: (*c* ≠ *Black*) = (*c* = *Red*)
by (*cases c*) *auto*

lemma *rbt_height_bheight_if*: *invc t* ⟹ *invh t* ⟹
height t ≤ (*if color t* = *Black then* 2 ∗ *bheight t else* 2 ∗ *bheight t* + *1*)
by(*induction t*) (*auto split*: *if_split_asm*)

lemma *rbt_height_bheight*: *rbt t* ⟹ *height t* / 2 ≤ *bheight t*
by(*auto simp*: *rbt_def dest*: *rbt_height_bheight_if*)

lemma *bheight_size_bound*: *invc t* ⟹ *invh t* ⟹ 2 ^ (*bheight t*) ≤ *size1 t*
by (*induction t*) *auto*

lemma *rbt_height_le*: **assumes** *rbt t* **shows** *height t* ≤ 2 ∗ *log* 2 (*size1 t*)
proof −
have 2 *powr* (*height t* / 2) ≤ 2 *powr bheight t*
using *rbt_height_bheight*[*OF assms*] **by** (*simp*)
also have ... ≤ *size1 t* **using** *assms*
by (*simp add*: *powr_realpow bheight_size_bound rbt_def*)
finally have 2 *powr* (*height t* / 2) ≤ *size1 t* **.**
hence *height t* / 2 ≤ *log* 2 (*size1 t*)
by (*simp add*: *le_log_iff size1_size del*: *divide_le_eq_numeral1*(*1*))
thus *?thesis* **by** *simp*
qed

end

17 Red-Black Tree Implementation of Maps

theory *RBT_Map*
imports
RBT_Set
Lookup2
begin

fun *upd* :: *'a*::*linorder* ⇒ *'b* ⇒ (*'a*∗*'b*) *rbt* ⇒ (*'a*∗*'b*) *rbt* **where**
upd x y Leaf = *R Leaf* (*x*,*y*) *Leaf* |
upd x y (*B l* (*a*,*b*) *r*) = (*case cmp x a of*
LT ⇒ *baliL* (*upd x y l*) (*a*,*b*) *r* |

```
  GT ⇒ baliR l (a,b) (upd x y r) |
  EQ ⇒ B l (x,y) r) |
upd x y (R l (a,b) r) = (case cmp x a of
  LT ⇒ R (upd x y l) (a,b) r |
  GT ⇒ R l (a,b) (upd x y r) |
  EQ ⇒ R l (x,y) r)

definition update :: 'a::linorder ⇒ 'b ⇒ ('a*'b) rbt ⇒ ('a*'b) rbt where
update x y t = paint Black (upd x y t)

fun del :: 'a::linorder ⇒ ('a*'b)rbt ⇒ ('a*'b)rbt where
del x Leaf = Leaf |
del x (Node l (a,b) c r) = (case cmp x a of
    LT ⇒ if l ≠ Leaf ∧ color l = Black
         then baldL (del x l) (a,b) r else R (del x l) (a,b) r |
    GT ⇒ if r ≠ Leaf∧ color r = Black
         then baldR l (a,b) (del x r) else R l (a,b) (del x r) |
  EQ ⇒ combine l r)

definition delete :: 'a::linorder ⇒ ('a*'b) rbt ⇒ ('a*'b) rbt where
delete x t = paint Black (del x t)
```

17.1 Functional Correctness Proofs

```
lemma inorder_upd:
  sorted1(inorder t) ⟹ inorder(upd x y t) = upd_list x y (inorder t)
by(induction x y t rule: upd.induct)
  (auto simp: upd_list_simps inorder_baliL inorder_baliR)

lemma inorder_update:
  sorted1(inorder t) ⟹ inorder(update x y t) = upd_list x y (inorder t)
by(simp add: update_def inorder_upd inorder_paint)

lemma inorder_del:
 sorted1(inorder t) ⟹  inorder(del x t) = del_list x (inorder t)
by(induction x t rule: del.induct)
  (auto simp: del_list_simps inorder_combine inorder_baldL inorder_baldR)

lemma inorder_delete:
  sorted1(inorder t) ⟹ inorder(delete x t) = del_list x (inorder t)
by(simp add: delete_def inorder_del inorder_paint)
```

17.2 Structural invariants

17.2.1 Update

lemma *invc_upd*: **assumes** *invc t*
shows *color t = Black ⟹ invc (upd x y t) invc2 (upd x y t)*
using *assms*
by (*induct x y t rule*: *upd.induct*) (*auto simp*: *invc_baliL invc_baliR invc2I*)

lemma *invh_upd*: **assumes** *invh t*
shows *invh (upd x y t) bheight (upd x y t) = bheight t*
using *assms*
by(*induct x y t rule*: *upd.induct*)
(*auto simp*: *invh_baliL invh_baliR bheight_baliL bheight_baliR*)

theorem *rbt_update*: *rbt t ⟹ rbt (update x y t)*
by (*simp add*: *invc_upd*(*2*) *invh_upd*(*1*) *color_paint_Black invc_paint_Black invh_paint*
rbt_def update_def)

17.2.2 Deletion

lemma *del_invc_invh*: *invh t ⟹ invc t ⟹ invh (del x t) ∧*
(color t = Red ∧ bheight (del x t) = bheight t ∧ invc (del x t) ∨
color t = Black ∧ bheight (del x t) = bheight t − 1 ∧ invc2 (del x t))
proof (*induct x t rule*: *del.induct*)
case (*2 x _ y _ c*)
have *x = y ∨ x < y ∨ x > y* **by** *auto*
thus *?case* **proof** (*elim disjE*)
assume *x = y*
with *2* **show** *?thesis*
by (*cases c*) (*simp_all add*: *invh_combine invc_combine*)
next
assume *x < y*
with *2* **show** *?thesis*
by(*cases c*)
(*auto simp*: *invh_baldL_invc invc_baldL invc2_baldL dest*: *neq_LeafD*)
next
assume *y < x*
with *2* **show** *?thesis*
by(*cases c*)
(*auto simp*: *invh_baldR_invc invc_baldR invc2_baldR dest*: *neq_LeafD*)
qed
qed *auto*

theorem *rbt_delete*: *rbt t* $\Longrightarrow$ *rbt* (*delete k t*)
by (*metis delete_def rbt_def color_paint_Black del_invc_invh invc_paint_Black invc2I invh_paint*)

interpretation *M*: *Map_by_Ordered*
where *empty* = *empty* **and** *lookup* = *lookup* **and** *update* = *update* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = *rbt*
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** (*simp add*: *empty_def*)
next
  case *2* **thus** *?case* **by**(*simp add*: *lookup_map_of*)
next
  case *3* **thus** *?case* **by**(*simp add*: *inorder_update*)
next
  case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
  case *5* **thus** *?case* **by** (*simp add*: *rbt_def empty_def*)
next
  case *6* **thus** *?case* **by** (*simp add*: *rbt_update*)
next
  case *7* **thus** *?case* **by** (*simp add*: *rbt_delete*)
qed

end

18 2-3 Trees

theory *Tree23*
imports *Main*
begin

class *height* =
fixes *height* :: *'a* $\Rightarrow$ *nat*

datatype *'a tree23* =
  Leaf (⟨⟩) |
  Node2 'a tree23 'a 'a tree23 (⟨_, _, _⟩) |
  Node3 'a tree23 'a 'a tree23 'a 'a tree23 (⟨_, _, _, _, _⟩)

fun *inorder* :: *'a tree23* $\Rightarrow$ *'a list* **where**
inorder Leaf = [] |
inorder(*Node2 l a r*) = *inorder l* @ *a* # *inorder r* |

inorder(*Node3 l a m b r*) = *inorder l* @ *a* # *inorder m* @ *b* # *inorder r*

instantiation *tree23* :: (*type*)*height*
begin

fun *height_tree23* :: *'a tree23* ⇒ *nat* **where**
height Leaf = *0* |
height (*Node2 l _ r*) = *Suc*(*max* (*height l*) (*height r*)) |
height (*Node3 l _ m _ r*) = *Suc*(*max* (*height l*) (*max* (*height m*) (*height r*)))

instance ..

end

Balanced:

fun *bal* :: *'a tree23* ⇒ *bool* **where**
bal Leaf = *True* |
bal (*Node2 l _ r*) = (*bal l* & *bal r* & *height l* = *height r*) |
bal (*Node3 l _ m _ r*) =
(*bal l* & *bal m* & *bal r* & *height l* = *height m* & *height m* = *height r*)

lemma *ht_sz_if_bal*: *bal t* ⟹ *2* ^ *height t* ≤ *size t* + *1*
by (*induction t*) *auto*

end

19 2-3 Tree Implementation of Sets

theory *Tree23_Set*
imports
Tree23
Cmp
Set_Specs
begin

declare *sorted_wrt.simps*(*2*)[*simp del*]

definition *empty* :: *'a tree23* **where**
empty = *Leaf*

fun *isin* :: *'a*::*linorder tree23* ⇒ *'a* ⇒ *bool* **where**
isin Leaf x = *False* |
isin (*Node2 l a r*) *x* =

```
  (case cmp x a of
     LT ⇒ isin l x |
     EQ ⇒ True |
     GT ⇒ isin r x) |
isin (Node3 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))

datatype 'a up_i = T_i 'a tree23 | Up_i 'a tree23 'a 'a tree23

fun tree_i :: 'a up_i ⇒ 'a tree23 where
tree_i (T_i t) = t |
tree_i (Up_i l a r) = Node2 l a r

fun ins :: 'a::linorder ⇒ 'a tree23 ⇒ 'a up_i where
ins x Leaf = Up_i Leaf x Leaf |
ins x (Node2 l a r) =
   (case cmp x a of
      LT ⇒
        (case ins x l of
           T_i l' => T_i (Node2 l' a r) |
           Up_i l1 b l2 => T_i (Node3 l1 b l2 a r)) |
      EQ ⇒ T_i (Node2 l x r) |
      GT ⇒
        (case ins x r of
           T_i r' => T_i (Node2 l a r') |
           Up_i r1 b r2 => T_i (Node3 l a r1 b r2))) |
ins x (Node3 l a m b r) =
   (case cmp x a of
      LT ⇒
        (case ins x l of
           T_i l' => T_i (Node3 l' a m b r) |
           Up_i l1 c l2 => Up_i (Node2 l1 c l2) a (Node2 m b r)) |
      EQ ⇒ T_i (Node3 l a m b r) |
      GT ⇒
        (case cmp x b of
           GT ⇒
             (case ins x r of
```

```
    T_i r' => T_i (Node3 l a m b r') |
    Up_i r1 c r2 => Up_i (Node2 l a m) b (Node2 r1 c r2)) |
  EQ ⇒ T_i (Node3 l a m b r) |
  LT ⇒
   (case ins x m of
    T_i m' => T_i (Node3 l a m' b r) |
    Up_i m1 c m2 => Up_i (Node2 l a m1) c (Node2 m2 b r))))
```

hide_const *insert*

definition *insert* :: *'a::linorder* ⇒ *'a tree23* ⇒ *'a tree23* **where**
insert x t = $tree_i$(*ins x t*)

datatype *'a* up_d = T_d *'a tree23* | Up_d *'a tree23*

fun $tree_d$ :: *'a* up_d ⇒ *'a tree23* **where**
$tree_d$ (T_d *t*) = *t* |
$tree_d$ (Up_d *t*) = *t*

fun *node21* :: *'a* up_d ⇒ *'a* ⇒ *'a tree23* ⇒ *'a* up_d **where**
node21 (T_d *t1*) *a t2* = T_d(*Node2 t1 a t2*) |
node21 (Up_d *t1*) *a* (*Node2 t2 b t3*) = Up_d(*Node3 t1 a t2 b t3*) |
node21 (Up_d *t1*) *a* (*Node3 t2 b t3 c t4*) = T_d(*Node2* (*Node2 t1 a t2*) *b* (*Node2 t3 c t4*))

fun *node22* :: *'a tree23* ⇒ *'a* ⇒ *'a* up_d ⇒ *'a* up_d **where**
node22 t1 a (T_d *t2*) = T_d(*Node2 t1 a t2*) |
node22 (*Node2 t1 b t2*) *a* (Up_d *t3*) = Up_d(*Node3 t1 b t2 a t3*) |
node22 (*Node3 t1 b t2 c t3*) *a* (Up_d *t4*) = T_d(*Node2* (*Node2 t1 b t2*) *c* (*Node2 t3 a t4*))

fun *node31* :: *'a* up_d ⇒ *'a* ⇒ *'a tree23* ⇒ *'a* ⇒ *'a tree23* ⇒ *'a* up_d **where**
node31 (T_d *t1*) *a t2 b t3* = T_d(*Node3 t1 a t2 b t3*) |
node31 (Up_d *t1*) *a* (*Node2 t2 b t3*) *c t4* = T_d(*Node2* (*Node3 t1 a t2 b t3*) *c t4*) |
node31 (Up_d *t1*) *a* (*Node3 t2 b t3 c t4*) *d t5* = T_d(*Node3* (*Node2 t1 a t2*) *b* (*Node2 t3 c t4*) *d t5*)

fun *node32* :: *'a tree23* ⇒ *'a* ⇒ *'a* up_d ⇒ *'a* ⇒ *'a tree23* ⇒ *'a* up_d **where**
node32 t1 a (T_d *t2*) *b t3* = T_d(*Node3 t1 a t2 b t3*) |
node32 t1 a (Up_d *t2*) *b* (*Node2 t3 c t4*) = T_d(*Node2 t1 a* (*Node3 t2 b t3 c t4*)) |

```
node32 t1 a (Up_d t2) b (Node3 t3 c t4 d t5) = T_d(Node3 t1 a (Node2 t2 b t3) c (Node2 t4 d t5))

fun node33 :: 'a tree23 ⇒ 'a ⇒ 'a tree23 ⇒ 'a ⇒ 'a up_d ⇒ 'a up_d where
node33 l a m b (T_d r) = T_d(Node3 l a m b r) |
node33 t1 a (Node2 t2 b t3) c (Up_d t4) = T_d(Node2 t1 a (Node3 t2 b t3 c t4)) |
node33 t1 a (Node3 t2 b t3 c t4) d (Up_d t5) = T_d(Node3 t1 a (Node2 t2 b t3) c (Node2 t4 d t5))

fun split_min :: 'a tree23 ⇒ 'a * 'a up_d where
split_min (Node2 Leaf a Leaf) = (a, Up_d Leaf) |
split_min (Node3 Leaf a Leaf b Leaf) = (a, T_d(Node2 Leaf b Leaf)) |
split_min (Node2 l a r) = (let (x,l') = split_min l in (x, node21 l' a r)) |
split_min (Node3 l a m b r) = (let (x,l') = split_min l in (x, node31 l' a m b r))
```

In the base cases of *split_min* and *del* it is enough to check if one subtree is a *Leaf*, in which case balancedness implies that so are the others. Exercise.

```
fun del :: 'a::linorder ⇒ 'a tree23 ⇒ 'a up_d where
del x Leaf = T_d Leaf |
del x (Node2 Leaf a Leaf) =
  (if x = a then Up_d Leaf else T_d(Node2 Leaf a Leaf)) |
del x (Node3 Leaf a Leaf b Leaf) =
  T_d(if x = a then Node2 Leaf b Leaf else
    if x = b then Node2 Leaf a Leaf
    else Node3 Leaf a Leaf b Leaf) |
del x (Node2 l a r) =
  (case cmp x a of
    LT ⇒ node21 (del x l) a r |
    GT ⇒ node22 l a (del x r) |
    EQ ⇒ let (a',t) = split_min r in node22 l a' t) |
del x (Node3 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)))

definition delete :: 'a::linorder ⇒ 'a tree23 ⇒ 'a tree23 where
delete x t = tree_d(del x t)
```

19.1 Functional Correctness

19.1.1 Proofs for isin

lemma *isin_set*: $sorted(inorder\ t) \Longrightarrow isin\ t\ x = (x \in set\ (inorder\ t))$
by (*induction t*) (*auto simp*: *isin_simps ball_Un*)

19.1.2 Proofs for insert

lemma *inorder_ins*:
$sorted(inorder\ t) \Longrightarrow inorder(tree_i(ins\ x\ t)) = ins\_list\ x\ (inorder\ t)$
by(*induction t*) (*auto simp*: *ins_list_simps split*: $up_i.splits$)

lemma *inorder_insert*:
$sorted(inorder\ t) \Longrightarrow inorder(insert\ a\ t) = ins\_list\ a\ (inorder\ t)$
by(*simp add*: *insert_def inorder_ins*)

19.1.3 Proofs for delete

lemma *inorder_node21*: $height\ r > 0 \Longrightarrow$
$inorder\ (tree_d\ (node21\ l'\ a\ r)) = inorder\ (tree_d\ l')\ @\ a\ \#\ inorder\ r$
by(*induct* l' *a r rule*: *node21.induct*) *auto*

lemma *inorder_node22*: $height\ l > 0 \Longrightarrow$
$inorder\ (tree_d\ (node22\ l\ a\ r')) = inorder\ l\ @\ a\ \#\ inorder\ (tree_d\ r')$
by(*induct l a* r' *rule*: *node22.induct*) *auto*

lemma *inorder_node31*: $height\ m > 0 \Longrightarrow$
$inorder\ (tree_d\ (node31\ l'\ a\ m\ b\ r)) = inorder\ (tree_d\ l')\ @\ a\ \#\ inorder\ m$
$@\ b\ \#\ inorder\ r$
by(*induct* l' *a m b r rule*: *node31.induct*) *auto*

lemma *inorder_node32*: $height\ r > 0 \Longrightarrow$
$inorder\ (tree_d\ (node32\ l\ a\ m'\ b\ r)) = inorder\ l\ @\ a\ \#\ inorder\ (tree_d\ m')$
$@\ b\ \#\ inorder\ r$
by(*induct l a* m' *b r rule*: *node32.induct*) *auto*

lemma *inorder_node33*: $height\ m > 0 \Longrightarrow$
$inorder\ (tree_d\ (node33\ l\ a\ m\ b\ r')) = inorder\ l\ @\ a\ \#\ inorder\ m\ @\ b\ \#$
$inorder\ (tree_d\ r')$
by(*induct l a m b* r' *rule*: *node33.induct*) *auto*

lemmas *inorder_nodes* = *inorder_node21 inorder_node22*
inorder_node31 inorder_node32 inorder_node33

lemma *split_minD*:
split_min t = (*x*,*t'*) ⟹ *bal t* ⟹ *height t* > *0* ⟹
x # *inorder*(*tree*$_d$ *t'*) = *inorder t*
by(*induction t arbitrary*: *t'* *rule*: *split_min.induct*)
(*auto simp*: *inorder_nodes split*: *prod.splits*)

lemma *inorder_del*: ⟦ *bal t* ; *sorted*(*inorder t*) ⟧ ⟹
inorder(*tree*$_d$ (*del x t*)) = *del_list x* (*inorder t*)
by(*induction t rule*: *del.induct*)
(*auto simp*: *del_list_simps inorder_nodes split_minD split*!: *if_split prod.splits*)

lemma *inorder_delete*: ⟦ *bal t* ; *sorted*(*inorder t*) ⟧ ⟹
inorder(*delete x t*) = *del_list x* (*inorder t*)
by(*simp add*: *delete_def inorder_del*)

19.2 Balancedness

19.2.1 Proofs for insert

First a standard proof that *ins* preserves *bal*.

instantiation *up*$_i$ :: (*type*)*height*
begin

fun *height_up*$_i$ :: *'a up*$_i$ ⇒ *nat* **where**
height (*T*$_i$ *t*) = *height t* |
height (*Up*$_i$ *l a r*) = *height l*

instance ..

end

lemma *bal_ins*: *bal t* ⟹ *bal* (*tree*$_i$(*ins a t*)) ∧ *height*(*ins a t*) = *height t*
by (*induct t*) (*auto split*!: *if_split up*$_i$*.split*)

Now an alternative proof (by Brian Huffman) that runs faster because two properties (balance and height) are combined in one predicate.

inductive *full* :: *nat* ⇒ *'a tree23* ⇒ *bool* **where**
full 0 Leaf |
⟦*full n l*; *full n r*⟧ ⟹ *full* (*Suc n*) (*Node2 l p r*) |
⟦*full n l*; *full n m*; *full n r*⟧ ⟹ *full* (*Suc n*) (*Node3 l p m q r*)

inductive_cases *full_elims*:
full n Leaf
full n (*Node2 l p r*)

full n (Node3 l p m q r)

inductive_cases *full_0_elim*: *full 0 t*
inductive_cases *full_Suc_elim*: *full (Suc n) t*

lemma *full_0_iff* [*simp*]: *full 0 t* $\longleftrightarrow$ *t = Leaf*
by (*auto elim*: *full_0_elim intro*: *full.intros*)

lemma *full_Leaf_iff* [*simp*]: *full n Leaf* $\longleftrightarrow$ *n = 0*
by (*auto elim*: *full_elims intro*: *full.intros*)

lemma *full_Suc_Node2_iff* [*simp*]:
full (Suc n) (Node2 l p r) $\longleftrightarrow$ *full n l* $\wedge$ *full n r*
by (*auto elim*: *full_elims intro*: *full.intros*)

lemma *full_Suc_Node3_iff* [*simp*]:
full (Suc n) (Node3 l p m q r) $\longleftrightarrow$ *full n l* $\wedge$ *full n m* $\wedge$ *full n r*
by (*auto elim*: *full_elims intro*: *full.intros*)

lemma *full_imp_height*: *full n t* $\Longrightarrow$ *height t = n*
by (*induct set*: *full*, *simp_all*)

lemma *full_imp_bal*: *full n t* $\Longrightarrow$ *bal t*
by (*induct set*: *full*, *auto dest*: *full_imp_height*)

lemma *bal_imp_full*: *bal t* $\Longrightarrow$ *full* (*height t*) *t*
by (*induct t*, *simp_all*)

lemma *bal_iff_full*: *bal t* $\longleftrightarrow$ ($\exists$ *n*. *full n t*)
by (*auto elim*!: *bal_imp_full full_imp_bal*)

The *insert* function either preserves the height of the tree, or increases it by one. The constructor returned by the *insert* function determines which: A return value of the form $T_i\ t$ indicates that the height will be the same. A value of the form $Up_i\ l\ p\ r$ indicates an increase in height.

fun $full_i$ :: *nat* $\Rightarrow$ *'a* up_i $\Rightarrow$ *bool* **where**
$full_i\ n\ (T_i\ t) \longleftrightarrow full\ n\ t$ |
$full_i\ n\ (Up_i\ l\ p\ r) \longleftrightarrow full\ n\ l \wedge full\ n\ r$

lemma $full_i\_ins$: *full n t* $\Longrightarrow$ $full_i\ n\ (ins\ a\ t)$
by (*induct rule*: *full.induct*) (*auto split*: $up_i.split$)

The *insert* operation preserves balance.

lemma *bal_insert*: *bal t* $\Longrightarrow$ *bal* (*insert a t*)

unfolding *bal_iff_full insert_def*
apply (*erule exE*)
apply (*drule full$_i$_ins* [*of* _ _ *a*])
apply (*cases ins a t*)
apply (*auto intro*: *full.intros*)
done

19.3 Proofs for delete

instantiation $up_d :: (type)height$
begin

fun *height_up$_d$* :: $'a\ up_d \Rightarrow nat$ **where**
height (T_d *t*) = *height t* |
height (Up_d *t*) = *height t* + *1*

instance ..

end

lemma *bal_tree$_d$_node21*:
⟦*bal r*; *bal* (*tree$_d$ l'*); *height r* = *height l'* ⟧ ⟹ *bal* (*tree$_d$* (*node21 l' a r*))
by(*induct l' a r rule*: *node21.induct*) *auto*

lemma *bal_tree$_d$_node22*:
⟦*bal*(*tree$_d$ r'*); *bal l*; *height r'* = *height l* ⟧ ⟹ *bal* (*tree$_d$* (*node22 l a r'*))
by(*induct l a r' rule*: *node22.induct*) *auto*

lemma *bal_tree$_d$_node31*:
⟦ *bal* (*tree$_d$ l'*); *bal m*; *bal r*; *height l'* = *height r*; *height m* = *height r* ⟧
⟹ *bal* (*tree$_d$* (*node31 l' a m b r*))
by(*induct l' a m b r rule*: *node31.induct*) *auto*

lemma *bal_tree$_d$_node32*:
⟦ *bal l*; *bal* (*tree$_d$ m'*); *bal r*; *height l* = *height r*; *height m'* = *height r* ⟧
⟹ *bal* (*tree$_d$* (*node32 l a m' b r*))
by(*induct l a m' b r rule*: *node32.induct*) *auto*

lemma *bal_tree$_d$_node33*:
⟦ *bal l*; *bal m*; *bal*(*tree$_d$ r'*); *height l* = *height r'*; *height m* = *height r'* ⟧
⟹ *bal* (*tree$_d$* (*node33 l a m b r'*))
by(*induct l a m b r' rule*: *node33.induct*) *auto*

lemmas *bals* = *bal_tree$_d$_node21 bal_tree$_d$_node22*

bal_tree$_d$_node31 bal_tree$_d$_node32 bal_tree$_d$_node33

lemma *height′_node21*:
height r > 0 ⟹ height(node21 l′ a r) = max (height l′) (height r) + 1
by(*induct l′ a r rule*: *node21.induct*)(*simp_all*)

lemma *height′_node22*:
height l > 0 ⟹ height(node22 l a r′) = max (height l) (height r′) + 1
by(*induct l a r′ rule*: *node22.induct*)(*simp_all*)

lemma *height′_node31*:
height m > 0 ⟹ height(node31 l a m b r) =
max (height l) (max (height m) (height r)) + 1
by(*induct l a m b r rule*: *node31.induct*)(*simp_all add*: *max_def*)

lemma *height′_node32*:
height r > 0 ⟹ height(node32 l a m b r) =
max (height l) (max (height m) (height r)) + 1
by(*induct l a m b r rule*: *node32.induct*)(*simp_all add*: *max_def*)

lemma *height′_node33*:
height m > 0 ⟹ height(node33 l a m b r) =
max (height l) (max (height m) (height r)) + 1
by(*induct l a m b r rule*: *node33.induct*)(*simp_all add*: *max_def*)

lemmas *heights = height′_node21 height′_node22*
height′_node31 height′_node32 height′_node33

lemma *height_split_min*:
split_min t = (x, t′) ⟹ height t > 0 ⟹ bal t ⟹ height t′ = height t
by(*induct t arbitrary*: *x t′ rule*: *split_min.induct*)
(*auto simp*: *heights split*: *prod.splits*)

lemma *height_del*: *bal t ⟹ height(del x t) = height t*
by(*induction x t rule*: *del.induct*)
(*auto simp*: *heights max_def height_split_min split*: *prod.splits*)

lemma *bal_split_min*:
⟦ *split_min t = (x, t′)*; *bal t*; *height t > 0* ⟧ *⟹ bal (tree$_d$ t′)*
by(*induct t arbitrary*: *x t′ rule*: *split_min.induct*)
(*auto simp*: *heights height_split_min bals split*: *prod.splits*)

lemma *bal_tree$_d$_del*: *bal t ⟹ bal(tree$_d$(del x t))*
by(*induction x t rule*: *del.induct*)

(*auto simp*: *bals bal_split_min height_del height_split_min split*: *prod.splits*)

corollary *bal_delete*: *bal t* $\Longrightarrow$ *bal*(*delete x t*)
by(*simp add*: *delete_def bal_tree*$_d$*_del*)

19.4 Overall Correctness

interpretation *S*: *Set_by_Ordered*
where *empty* = *empty* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = *bal*
proof (*standard*, *goal_cases*)
  case *2* **thus** *?case* **by**(*simp add*: *isin_set*)
next
  case *3* **thus** *?case* **by**(*simp add*: *inorder_insert*)
next
  case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
  case *6* **thus** *?case* **by**(*simp add*: *bal_insert*)
next
  case *7* **thus** *?case* **by**(*simp add*: *bal_delete*)
qed (*simp add*: *empty_def*)+

end

20 2-3 Tree Implementation of Maps

theory *Tree23_Map*
imports
  Tree23_Set
  Map_Specs
begin

fun *lookup* :: ('*a*::*linorder* * '*b*) *tree23* $\Rightarrow$ '*a* $\Rightarrow$ '*b option* **where**
lookup Leaf x = *None* |
lookup (*Node2 l* (*a*,*b*) *r*) *x* = (*case cmp x a of*
  LT $\Rightarrow$ *lookup l x* |
  GT $\Rightarrow$ *lookup r x* |
  EQ $\Rightarrow$ *Some b*) |
lookup (*Node3 l* (*a1*,*b1*) *m* (*a2*,*b2*) *r*) *x* = (*case cmp x a1 of*
  LT $\Rightarrow$ *lookup l x* |
  EQ $\Rightarrow$ *Some b1* |
  GT $\Rightarrow$ (*case cmp x a2 of*
      LT $\Rightarrow$ *lookup m x* |

```
        EQ ⇒ Some b2 |
        GT ⇒ lookup r x))

fun upd :: 'a::linorder ⇒ 'b ⇒ ('a*'b) tree23 ⇒ ('a*'b) up_i where
upd x y Leaf = Up_i Leaf (x,y) Leaf |
upd x y (Node2 l ab r) = (case cmp x (fst ab) of
  LT ⇒ (case upd x y l of
         T_i l' => T_i (Node2 l' ab r)
       | Up_i l1 ab' l2 => T_i (Node3 l1 ab' l2 ab r)) |
  EQ ⇒ T_i (Node2 l (x,y) r) |
  GT ⇒ (case upd x y r of
         T_i r' => T_i (Node2 l ab r')
       | Up_i r1 ab' r2 => T_i (Node3 l ab r1 ab' r2))) |
upd x y (Node3 l ab1 m ab2 r) = (case cmp x (fst ab1) of
  LT ⇒ (case upd x y l of
         T_i l' => T_i (Node3 l' ab1 m ab2 r)
       | Up_i l1 ab' l2 => Up_i (Node2 l1 ab' l2) ab1 (Node2 m ab2 r)) |
  EQ ⇒ T_i (Node3 l (x,y) m ab2 r) |
  GT ⇒ (case cmp x (fst ab2) of
         LT ⇒ (case upd x y m of
                T_i m' => T_i (Node3 l ab1 m' ab2 r)
              | Up_i m1 ab' m2 => Up_i (Node2 l ab1 m1) ab' (Node2 m2
ab2 r)) |
         EQ ⇒ T_i (Node3 l ab1 m (x,y) r) |
         GT ⇒ (case upd x y r of
                T_i r' => T_i (Node3 l ab1 m ab2 r')
              | Up_i r1 ab' r2 => Up_i (Node2 l ab1 m) ab2 (Node2 r1 ab'
r2))))

definition update :: 'a::linorder ⇒ 'b ⇒ ('a*'b) tree23 ⇒ ('a*'b) tree23
where
update a b t = tree_i(upd a b t)

fun del :: 'a::linorder ⇒ ('a*'b) tree23 ⇒ ('a*'b) up_d where
del x Leaf = T_d Leaf |
del x (Node2 Leaf ab1 Leaf) = (if x=fst ab1 then Up_d Leaf else T_d(Node2
Leaf ab1 Leaf)) |
del x (Node3 Leaf ab1 Leaf ab2 Leaf) = T_d(if x=fst ab1 then Node2 Leaf
ab2 Leaf
  else if x=fst ab2 then Node2 Leaf ab1 Leaf else Node3 Leaf ab1 Leaf ab2
Leaf) |
del x (Node2 l ab1 r) = (case cmp x (fst ab1) of
  LT ⇒ node21 (del x l) ab1 r |
  GT ⇒ node22 l ab1 (del x r) |
```

```
  EQ ⇒ let (ab1',t) = split_min r in node22 l ab1' t) |
del x (Node3 l ab1 m ab2 r) = (case cmp x (fst ab1) of
  LT ⇒ node31 (del x l) ab1 m ab2 r |
  EQ ⇒ let (ab1',m') = split_min m in node32 l ab1' m' ab2 r |
  GT ⇒ (case cmp x (fst ab2) of
          LT ⇒ node32 l ab1 (del x m) ab2 r |
          EQ ⇒ let (ab2',r') = split_min r in node33 l ab1 m ab2' r' |
          GT ⇒ node33 l ab1 m ab2 (del x r)))
```

definition *delete* :: *'a::linorder* ⇒ *('a*'b) tree23* ⇒ *('a*'b) tree23* **where**
delete x t = $tree_d$(*del x t*)

20.1 Functional Correctness

lemma *lookup_map_of*:
sorted1(*inorder t*) ⟹ *lookup t x* = *map_of* (*inorder t*) *x*
by (*induction t*) (*auto simp*: *map_of_simps split*: *option.split*)

lemma *inorder_upd*:
sorted1(*inorder t*) ⟹ *inorder*($tree_i$(*upd x y t*)) = *upd_list x y* (*inorder t*)
by(*induction t*) (*auto simp*: *upd_list_simps split*: up_i.*splits*)

corollary *inorder_update*:
sorted1(*inorder t*) ⟹ *inorder*(*update x y t*) = *upd_list x y* (*inorder t*)
by(*simp add*: *update_def inorder_upd*)

lemma *inorder_del*: ⟦ *bal t* ; *sorted1*(*inorder t*) ⟧ ⟹
inorder($tree_d$ (*del x t*)) = *del_list x* (*inorder t*)
by(*induction t rule*: *del.induct*)
(*auto simp*: *del_list_simps inorder_nodes split_minD split*!: *if_split prod.splits*)

corollary *inorder_delete*: ⟦ *bal t* ; *sorted1*(*inorder t*) ⟧ ⟹
inorder(*delete x t*) = *del_list x* (*inorder t*)
by(*simp add*: *delete_def inorder_del*)

20.2 Balancedness

lemma *bal_upd*: *bal t* ⟹ *bal* ($tree_i$(*upd x y t*)) ∧ *height*(*upd x y t*) = *height t*
by (*induct t*) (*auto split*!: *if_split* up_i.*split*)

corollary *bal_update*: *bal t* ⟹ *bal* (*update x y t*)

by (*simp add*: *update_def bal_upd*)

lemma *height_del*: *bal t* $\Longrightarrow$ *height*(*del x t*) = *height t*
by(*induction x t rule*: *del.induct*)
(*auto simp add*: *heights max_def height_split_min split*: *prod.split*)

lemma *bal_tree*$_d$*_del*: *bal t* $\Longrightarrow$ *bal*(*tree*$_d$(*del x t*))
by(*induction x t rule*: *del.induct*)
(*auto simp*: *bals bal_split_min height_del height_split_min split*: *prod.split*)

corollary *bal_delete*: *bal t* $\Longrightarrow$ *bal*(*delete x t*)
by(*simp add*: *delete_def bal_tree*$_d$*_del*)

20.3 Overall Correctness

interpretation *M*: *Map_by_Ordered*
where *empty* = *empty* **and** *lookup* = *lookup* **and** *update* = *update* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = *bal*
proof (*standard*, *goal_cases*)
case *1* **thus** *?case* **by**(*simp add*: *empty_def*)
next
case *2* **thus** *?case* **by**(*simp add*: *lookup_map_of*)
next
case *3* **thus** *?case* **by**(*simp add*: *inorder_update*)
next
case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
case *5* **thus** *?case* **by**(*simp add*: *empty_def*)
next
case *6* **thus** *?case* **by**(*simp add*: *bal_update*)
next
case *7* **thus** *?case* **by**(*simp add*: *bal_delete*)
qed

end

21 2-3-4 Trees

theory *Tree234*
imports *Main*
begin

class *height* =
fixes *height* :: *'a* ⇒ *nat*

datatype *'a tree234* =
Leaf (⟨⟩) |
Node2 'a tree234 'a 'a tree234 (⟨_, _, _⟩) |
Node3 'a tree234 'a 'a tree234 'a 'a tree234 (⟨_, _, _, _, _⟩) |
Node4 'a tree234 'a 'a tree234 'a 'a tree234 'a 'a tree234
(⟨_, _, _, _, _, _, _⟩)

fun *inorder* :: *'a tree234* ⇒ *'a list* **where**
inorder Leaf = [] |
inorder(*Node2 l a r*) = *inorder l* @ *a* # *inorder r* |
inorder(*Node3 l a m b r*) = *inorder l* @ *a* # *inorder m* @ *b* # *inorder r* |
inorder(*Node4 l a m b n c r*) = *inorder l* @ *a* # *inorder m* @ *b* # *inorder n* @ *c* # *inorder r*

instantiation *tree234* :: (*type*)*height*
begin

fun *height_tree234* :: *'a tree234* ⇒ *nat* **where**
height Leaf = *0* |
height (*Node2 l* _ *r*) = *Suc*(*max* (*height l*) (*height r*)) |
height (*Node3 l* _ *m* _ *r*) = *Suc*(*max* (*height l*) (*max* (*height m*) (*height r*))) |
height (*Node4 l* _ *m* _ *n* _ *r*) = *Suc*(*max* (*height l*) (*max* (*height m*) (*max* (*height n*) (*height r*))))

instance ..

end

Balanced:

fun *bal* :: *'a tree234* ⇒ *bool* **where**
bal Leaf = *True* |
bal (*Node2 l* _ *r*) = (*bal l* & *bal r* & *height l* = *height r*) |
bal (*Node3 l* _ *m* _ *r*) = (*bal l* & *bal m* & *bal r* & *height l* = *height m* & *height m* = *height r*) |
bal (*Node4 l* _ *m* _ *n* _ *r*) = (*bal l* & *bal m* & *bal n* & *bal r* & *height l* = *height m* & *height m* = *height n* & *height n* = *height r*)

end

22 2-3-4 Tree Implementation of Sets

theory *Tree234_Set*
imports
Tree234
Cmp
Set_Specs
begin

declare *sorted_wrt.simps*(*2*)[*simp del*]

22.1 Set operations on 2-3-4 trees

definition *empty* :: *'a tree234* **where**
empty = *Leaf*

fun *isin* :: *'a*::*linorder tree234* ⇒ *'a* ⇒ *bool* **where**
isin Leaf x = *False* |
isin (*Node2 l a r*) *x* =
(*case cmp x a of LT* ⇒ *isin l x* | *EQ* ⇒ *True* | *GT* ⇒ *isin r x*) |
isin (*Node3 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*)) |
isin (*Node4 t1 a t2 b t3 c t4*) *x* =
(*case cmp x b of*
LT ⇒
(*case cmp x a of*
LT ⇒ *isin t1 x* |
EQ ⇒ *True* |
GT ⇒ *isin t2 x*) |
EQ ⇒ *True* |
GT ⇒
(*case cmp x c of*
LT ⇒ *isin t3 x* |
EQ ⇒ *True* |
GT ⇒ *isin t4 x*))

datatype *'a up*$_i$ = *T*$_i$ *'a tree234* | *Up*$_i$ *'a tree234 'a 'a tree234*

fun *tree*$_i$ :: *'a up*$_i$ ⇒ *'a tree234* **where**
tree$_i$ (*T*$_i$ *t*) = *t* |
tree$_i$ (*Up*$_i$ *l a r*) = *Node2 l a r*

fun *ins* :: *'a*::*linorder* ⇒ *'a tree234* ⇒ *'a up*$_i$ **where**

```
ins x Leaf = Up_i Leaf x Leaf |
ins x (Node2 l a r) =
  (case cmp x a of
    LT ⇒ (case ins x l of
           T_i l' => T_i (Node2 l' a r)
         | Up_i l1 b l2 => T_i (Node3 l1 b l2 a r)) |
    EQ ⇒ T_i (Node2 l x r) |
    GT ⇒ (case ins x r of
           T_i r' => T_i (Node2 l a r')
         | Up_i r1 b r2 => T_i (Node3 l a r1 b r2))) |
ins x (Node3 l a m b r) =
  (case cmp x a of
    LT ⇒ (case ins x l of
           T_i l' => T_i (Node3 l' a m b r)
         | Up_i l1 c l2 => Up_i (Node2 l1 c l2) a (Node2 m b r)) |
    EQ ⇒ T_i (Node3 l a m b r) |
    GT ⇒ (case cmp x b of
           GT ⇒ (case ins x r of
                  T_i r' => T_i (Node3 l a m b r')
                | Up_i r1 c r2 => Up_i (Node2 l a m) b (Node2 r1 c r2)) |
           EQ ⇒ T_i (Node3 l a m b r) |
           LT ⇒ (case ins x m of
                  T_i m' => T_i (Node3 l a m' b r)
                | Up_i m1 c m2 => Up_i (Node2 l a m1) c (Node2 m2 b
r)))) |
ins x (Node4 t1 a t2 b t3 c t4) =
  (case cmp x b of
    LT ⇒
     (case cmp x a of
       LT ⇒
        (case ins x t1 of
           T_i t => T_i (Node4 t a t2 b t3 c t4) |
           Up_i l y r => Up_i (Node2 l y r) a (Node3 t2 b t3 c t4)) |
       EQ ⇒ T_i (Node4 t1 a t2 b t3 c t4) |
       GT ⇒
        (case ins x t2 of
           T_i t => T_i (Node4 t1 a t b t3 c t4) |
           Up_i l y r => Up_i (Node2 t1 a l) y (Node3 r b t3 c t4))) |
    EQ ⇒ T_i (Node4 t1 a t2 b t3 c t4) |
    GT ⇒
     (case cmp x c of
       LT ⇒
        (case ins x t3 of
          T_i t => T_i (Node4 t1 a t2 b t c t4) |
```

```
    Up_i l y r => Up_i (Node2 t1 a t2) b (Node3 l y r c t4)) |
  EQ ⇒ T_i (Node4 t1 a t2 b t3 c t4) |
  GT ⇒
   (case ins x t4 of
     T_i t => T_i (Node4 t1 a t2 b t3 c t) |
     Up_i l y r => Up_i (Node2 t1 a t2) b (Node3 t3 c l y r))))
```

hide_const *insert*

definition *insert* :: *'a::linorder* ⇒ *'a tree234* ⇒ *'a tree234* **where**
insert x t = *tree_i*(*ins x t*)

datatype *'a up_d* = *T_d 'a tree234* | *Up_d 'a tree234*

fun *tree_d* :: *'a up_d* ⇒ *'a tree234* **where**
tree_d (*T_d t*) = *t* |
tree_d (*Up_d t*) = *t*

fun *node21* :: *'a up_d* ⇒ *'a* ⇒ *'a tree234* ⇒ *'a up_d* **where**
node21 (*T_d l*) *a r* = *T_d*(*Node2 l a r*) |
node21 (*Up_d l*) *a* (*Node2 lr b rr*) = *Up_d*(*Node3 l a lr b rr*) |
node21 (*Up_d l*) *a* (*Node3 lr b mr c rr*) = *T_d*(*Node2* (*Node2 l a lr*) *b* (*Node2 mr c rr*)) |
node21 (*Up_d t1*) *a* (*Node4 t2 b t3 c t4 d t5*) = *T_d*(*Node2* (*Node2 t1 a t2*) *b* (*Node3 t3 c t4 d t5*))

fun *node22* :: *'a tree234* ⇒ *'a* ⇒ *'a up_d* ⇒ *'a up_d* **where**
node22 l a (*T_d r*) = *T_d*(*Node2 l a r*) |
node22 (*Node2 ll b rl*) *a* (*Up_d r*) = *Up_d*(*Node3 ll b rl a r*) |
node22 (*Node3 ll b ml c rl*) *a* (*Up_d r*) = *T_d*(*Node2* (*Node2 ll b ml*) *c* (*Node2 rl a r*)) |
node22 (*Node4 t1 a t2 b t3 c t4*) *d* (*Up_d t5*) = *T_d*(*Node2* (*Node2 t1 a t2*) *b* (*Node3 t3 c t4 d t5*))

fun *node31* :: *'a up_d* ⇒ *'a* ⇒ *'a tree234* ⇒ *'a* ⇒ *'a tree234* ⇒ *'a up_d* **where**
node31 (*T_d t1*) *a t2 b t3* = *T_d*(*Node3 t1 a t2 b t3*) |
node31 (*Up_d t1*) *a* (*Node2 t2 b t3*) *c t4* = *T_d*(*Node2* (*Node3 t1 a t2 b t3*) *c t4*) |
node31 (*Up_d t1*) *a* (*Node3 t2 b t3 c t4*) *d t5* = *T_d*(*Node3* (*Node2 t1 a t2*) *b* (*Node2 t3 c t4*) *d t5*) |
node31 (*Up_d t1*) *a* (*Node4 t2 b t3 c t4 d t5*) *e t6* = *T_d*(*Node3* (*Node2 t1 a t2*) *b* (*Node3 t3 c t4 d t5*) *e t6*)

fun *node32* :: *'a tree234* ⇒ *'a* ⇒ *'a up_d* ⇒ *'a* ⇒ *'a tree234* ⇒ *'a up_d* **where**

node32 t1 a (T_d *t2*) *b t3* = T_d(*Node3 t1 a t2 b t3*) |
node32 t1 a (Up_d *t2*) *b* (*Node2 t3 c t4*) = T_d(*Node2 t1 a* (*Node3 t2 b t3 c t4*)) |
node32 t1 a (Up_d *t2*) *b* (*Node3 t3 c t4 d t5*) = T_d(*Node3 t1 a* (*Node2 t2 b t3*) *c* (*Node2 t4 d t5*)) |
node32 t1 a (Up_d *t2*) *b* (*Node4 t3 c t4 d t5 e t6*) = T_d(*Node3 t1 a* (*Node2 t2 b t3*) *c* (*Node3 t4 d t5 e t6*))

fun *node33* :: *'a tree234* ⇒ *'a* ⇒ *'a tree234* ⇒ *'a* ⇒ *'a* up_d ⇒ *'a* up_d **where**
node33 l a m b (T_d *r*) = T_d(*Node3 l a m b r*) |
node33 t1 a (*Node2 t2 b t3*) *c* (Up_d *t4*) = T_d(*Node2 t1 a* (*Node3 t2 b t3 c t4*)) |
node33 t1 a (*Node3 t2 b t3 c t4*) *d* (Up_d *t5*) = T_d(*Node3 t1 a* (*Node2 t2 b t3*) *c* (*Node2 t4 d t5*)) |
node33 t1 a (*Node4 t2 b t3 c t4 d t5*) *e* (Up_d *t6*) = T_d(*Node3 t1 a* (*Node2 t2 b t3*) *c* (*Node3 t4 d t5 e t6*))

fun *node41* :: *'a* up_d ⇒ *'a* ⇒ *'a tree234* ⇒ *'a* ⇒ *'a tree234* ⇒ *'a* ⇒ *'a tree234* ⇒ *'a* up_d **where**
node41 (T_d *t1*) *a t2 b t3 c t4* = T_d(*Node4 t1 a t2 b t3 c t4*) |
node41 (Up_d *t1*) *a* (*Node2 t2 b t3*) *c t4 d t5* = T_d(*Node3* (*Node3 t1 a t2 b t3*) *c t4 d t5*) |
node41 (Up_d *t1*) *a* (*Node3 t2 b t3 c t4*) *d t5 e t6* = T_d(*Node4* (*Node2 t1 a t2*) *b* (*Node2 t3 c t4*) *d t5 e t6*) |
node41 (Up_d *t1*) *a* (*Node4 t2 b t3 c t4 d t5*) *e t6 f t7* = T_d(*Node4* (*Node2 t1 a t2*) *b* (*Node3 t3 c t4 d t5*) *e t6 f t7*)

fun *node42* :: *'a tree234* ⇒ *'a* ⇒ *'a* up_d ⇒ *'a* ⇒ *'a tree234* ⇒ *'a* ⇒ *'a tree234* ⇒ *'a* up_d **where**
node42 t1 a (T_d *t2*) *b t3 c t4* = T_d(*Node4 t1 a t2 b t3 c t4*) |
node42 (*Node2 t1 a t2*) *b* (Up_d *t3*) *c t4 d t5* = T_d(*Node3* (*Node3 t1 a t2 b t3*) *c t4 d t5*) |
node42 (*Node3 t1 a t2 b t3*) *c* (Up_d *t4*) *d t5 e t6* = T_d(*Node4* (*Node2 t1 a t2*) *b* (*Node2 t3 c t4*) *d t5 e t6*) |
node42 (*Node4 t1 a t2 b t3 c t4*) *d* (Up_d *t5*) *e t6 f t7* = T_d(*Node4* (*Node2 t1 a t2*) *b* (*Node3 t3 c t4 d t5*) *e t6 f t7*)

fun *node43* :: *'a tree234* ⇒ *'a* ⇒ *'a tree234* ⇒ *'a* ⇒ *'a* up_d ⇒ *'a* ⇒ *'a tree234* ⇒ *'a* up_d **where**
node43 t1 a t2 b (T_d *t3*) *c t4* = T_d(*Node4 t1 a t2 b t3 c t4*) |
node43 t1 a (*Node2 t2 b t3*) *c* (Up_d *t4*) *d t5* = T_d(*Node3 t1 a* (*Node3 t2 b t3 c t4*) *d t5*) |
node43 t1 a (*Node3 t2 b t3 c t4*) *d* (Up_d *t5*) *e t6* = T_d(*Node4 t1 a* (*Node2 t2 b t3*) *c* (*Node2 t4 d t5*) *e t6*) |

node43 t1 a (Node4 t2 b t3 c t4 d t5) e (Up$_d$ t6) f t7 = T$_d$(Node4 t1 a (Node2 t2 b t3) c (Node3 t4 d t5 e t6) f t7)

fun *node44 :: 'a tree234 ⇒ 'a ⇒ 'a tree234 ⇒ 'a ⇒ 'a tree234 ⇒ 'a ⇒ 'a up$_d$ ⇒ 'a up$_d$* **where**
node44 t1 a t2 b t3 c (T$_d$ t4) = T$_d$(Node4 t1 a t2 b t3 c t4) |
node44 t1 a t2 b (Node2 t3 c t4) d (Up$_d$ t5) = T$_d$(Node3 t1 a t2 b (Node3 t3 c t4 d t5)) |
node44 t1 a t2 b (Node3 t3 c t4 d t5) e (Up$_d$ t6) = T$_d$(Node4 t1 a t2 b (Node2 t3 c t4) d (Node2 t5 e t6)) |
node44 t1 a t2 b (Node4 t3 c t4 d t5 e t6) f (Up$_d$ t7) = T$_d$(Node4 t1 a t2 b (Node2 t3 c t4) d (Node3 t5 e t6 f t7))

fun *split_min :: 'a tree234 ⇒ 'a ∗ 'a up$_d$* **where**
split_min (Node2 Leaf a Leaf) = (a, Up$_d$ Leaf) |
split_min (Node3 Leaf a Leaf b Leaf) = (a, T$_d$(Node2 Leaf b Leaf)) |
split_min (Node4 Leaf a Leaf b Leaf c Leaf) = (a, T$_d$(Node3 Leaf b Leaf c Leaf)) |
split_min (Node2 l a r) = (let (x,l') = split_min l in (x, node21 l' a r)) |
split_min (Node3 l a m b r) = (let (x,l') = split_min l in (x, node31 l' a m b r)) |
split_min (Node4 l a m b n c r) = (let (x,l') = split_min l in (x, node41 l' a m b n c r))

fun *del :: 'a::linorder ⇒ 'a tree234 ⇒ 'a up$_d$* **where**
del k Leaf = T$_d$ Leaf |
del k (Node2 Leaf p Leaf) = (if k=p then Up$_d$ Leaf else T$_d$(Node2 Leaf p Leaf)) |
del k (Node3 Leaf p Leaf q Leaf) = T$_d$(if k=p then Node2 Leaf q Leaf
* else if k=q then Node2 Leaf p Leaf else Node3 Leaf p Leaf q Leaf) |*
del k (Node4 Leaf a Leaf b Leaf c Leaf) =
* T$_d$(if k=a then Node3 Leaf b Leaf c Leaf else*
*   if k=b then Node3 Leaf a Leaf c Leaf else*
*   if k=c then Node3 Leaf a Leaf b Leaf*
*   else Node4 Leaf a Leaf b Leaf c Leaf) |*
del k (Node2 l a r) = (case cmp k a of
* LT ⇒ node21 (del k l) a r |*
* GT ⇒ node22 l a (del k r) |*
* EQ ⇒ let (a',t) = split_min r in node22 l a' t) |*
del k (Node3 l a m b r) = (case cmp k a of
* LT ⇒ node31 (del k l) a m b r |*
* EQ ⇒ let (a',m') = split_min m in node32 l a' m' b r |*
* GT ⇒ (case cmp k b of*
*   LT ⇒ node32 l a (del k m) b r |*

```
        EQ ⇒ let (b',r') = split_min r in node33 l a m b' r' |
        GT ⇒ node33 l a m b (del k r))) |
del k (Node4 l a m b n c r) = (case cmp k b of
  LT ⇒ (case cmp k a of
        LT ⇒ node41 (del k l) a m b n c r |
        EQ ⇒ let (a',m') = split_min m in node42 l a' m' b n c r |
        GT ⇒ node42 l a (del k m) b n c r) |
  EQ ⇒ let (b',n') = split_min n in node43 l a m b' n' c r |
  GT ⇒ (case cmp k c of
        LT ⇒ node43 l a m b (del k n) c r |
        EQ ⇒ let (c',r') = split_min r in node44 l a m b n c' r' |
        GT ⇒ node44 l a m b n c (del k r)))
```

definition *delete* :: *'a::linorder* ⇒ *'a tree234* ⇒ *'a tree234* **where**
delete x t = $tree_d$(*del x t*)

22.2 Functional correctness

22.2.1 Functional correctness of isin:

lemma *isin_set*: *sorted*(*inorder t*) ⟹ *isin t x* = (*x* ∈ *set* (*inorder t*))
by (*induction t*) (*auto simp*: *isin_simps ball_Un*)

22.2.2 Functional correctness of insert:

lemma *inorder_ins*:
  sorted(*inorder t*) ⟹ *inorder*($tree_i$(*ins x t*)) = *ins_list x* (*inorder t*)
by(*induction t*) (*auto*, *auto simp*: *ins_list_simps split*!: *if_splits* up_i.*splits*)

lemma *inorder_insert*:
  sorted(*inorder t*) ⟹ *inorder*(*insert a t*) = *ins_list a* (*inorder t*)
by(*simp add*: *insert_def inorder_ins*)

22.2.3 Functional correctness of delete

lemma *inorder_node21*: *height r* > *0* ⟹
  inorder ($tree_d$ (*node21 l' a r*)) = *inorder* ($tree_d$ *l'*) @ *a* # *inorder r*
by(*induct l' a r rule*: *node21.induct*) *auto*

lemma *inorder_node22*: *height l* > *0* ⟹
  inorder ($tree_d$ (*node22 l a r'*)) = *inorder l* @ *a* # *inorder* ($tree_d$ *r'*)
by(*induct l a r' rule*: *node22.induct*) *auto*

lemma *inorder_node31*: *height m* > *0* ⟹

inorder ($tree_d$ (*node31* l' a m b r)) = *inorder* ($tree_d$ l') @ a # *inorder* m @ b # *inorder* r
by(*induct* l' a m b r *rule*: *node31.induct*) *auto*

lemma *inorder_node32*: *height* $r > 0 \Longrightarrow$
inorder ($tree_d$ (*node32* l a m' b r)) = *inorder* l @ a # *inorder* ($tree_d$ m') @ b # *inorder* r
by(*induct* l a m' b r *rule*: *node32.induct*) *auto*

lemma *inorder_node33*: *height* $m > 0 \Longrightarrow$
inorder ($tree_d$ (*node33* l a m b r')) = *inorder* l @ a # *inorder* m @ b # *inorder* ($tree_d$ r')
by(*induct* l a m b r' *rule*: *node33.induct*) *auto*

lemma *inorder_node41*: *height* $m > 0 \Longrightarrow$
inorder ($tree_d$ (*node41* l' a m b n c r)) = *inorder* ($tree_d$ l') @ a # *inorder* m @ b # *inorder* n @ c # *inorder* r
by(*induct* l' a m b n c r *rule*: *node41.induct*) *auto*

lemma *inorder_node42*: *height* $l > 0 \Longrightarrow$
inorder ($tree_d$ (*node42* l a m b n c r)) = *inorder* l @ a # *inorder* ($tree_d$ m) @ b # *inorder* n @ c # *inorder* r
by(*induct* l a m b n c r *rule*: *node42.induct*) *auto*

lemma *inorder_node43*: *height* $m > 0 \Longrightarrow$
inorder ($tree_d$ (*node43* l a m b n c r)) = *inorder* l @ a # *inorder* m @ b # *inorder*($tree_d$ n) @ c # *inorder* r
by(*induct* l a m b n c r *rule*: *node43.induct*) *auto*

lemma *inorder_node44*: *height* $n > 0 \Longrightarrow$
inorder ($tree_d$ (*node44* l a m b n c r)) = *inorder* l @ a # *inorder* m @ b # *inorder* n @ c # *inorder* ($tree_d$ r)
by(*induct* l a m b n c r *rule*: *node44.induct*) *auto*

lemmas *inorder_nodes* = *inorder_node21 inorder_node22*
inorder_node31 inorder_node32 inorder_node33
inorder_node41 inorder_node42 inorder_node43 inorder_node44

lemma *split_minD*:
split_min t = (x,t') $\Longrightarrow$ *bal* t $\Longrightarrow$ *height* $t > 0 \Longrightarrow$
x # *inorder*($tree_d$ t') = *inorder* t
by(*induction* t *arbitrary*: t' *rule*: *split_min.induct*)
(*auto simp*: *inorder_nodes split*: *prod.splits*)

lemma *inorder_del*: ⟦ *bal t* ; *sorted*(*inorder t*) ⟧ ⟹
inorder(*tree*$_d$ (*del x t*)) = *del_list x* (*inorder t*)
by(*induction t rule*: *del.induct*)
(*auto simp*: *inorder_nodes del_list_simps split_minD split*!: *if_split prod.splits*)

lemma *inorder_delete*: ⟦ *bal t* ; *sorted*(*inorder t*) ⟧ ⟹
inorder(*delete x t*) = *del_list x* (*inorder t*)
by(*simp add*: *delete_def inorder_del*)

22.3 Balancedness

22.3.1 Proofs for insert

First a standard proof that *ins* preserves *bal*.

instantiation *up*$_i$ :: (*type*)*height*
begin

fun *height_up*$_i$ :: *'a up*$_i$ ⇒ *nat* **where**
height (*T*$_i$ *t*) = *height t* |
height (*Up*$_i$ *l a r*) = *height l*

instance ..

end

lemma *bal_ins*: *bal t* ⟹ *bal* (*tree*$_i$(*ins a t*)) ∧ *height*(*ins a t*) = *height t*
by (*induct t*) (*auto split*!: *if_split up*$_i$*.split*)

Now an alternative proof (by Brian Huffman) that runs faster because two properties (balance and height) are combined in one predicate.

inductive *full* :: *nat* ⇒ *'a tree234* ⇒ *bool* **where**
full 0 Leaf |
⟦*full n l*; *full n r*⟧ ⟹ *full* (*Suc n*) (*Node2 l p r*) |
⟦*full n l*; *full n m*; *full n r*⟧ ⟹ *full* (*Suc n*) (*Node3 l p m q r*) |
⟦*full n l*; *full n m*; *full n m'*; *full n r*⟧ ⟹ *full* (*Suc n*) (*Node4 l p m q m'* *q'* *r*)

inductive_cases *full_elims*:
full n Leaf
full n (*Node2 l p r*)
full n (*Node3 l p m q r*)
full n (*Node4 l p m q m' q' r*)

inductive_cases *full_0_elim*: *full 0 t*
inductive_cases *full_Suc_elim*: *full (Suc n) t*

lemma *full_0_iff* [*simp*]: *full 0 t* $\longleftrightarrow$ *t = Leaf*
by (*auto elim*: *full_0_elim intro*: *full.intros*)

lemma *full_Leaf_iff* [*simp*]: *full n Leaf* $\longleftrightarrow$ *n = 0*
by (*auto elim*: *full_elims intro*: *full.intros*)

lemma *full_Suc_Node2_iff* [*simp*]:
full (Suc n) (Node2 l p r) $\longleftrightarrow$ *full n l* $\wedge$ *full n r*
by (*auto elim*: *full_elims intro*: *full.intros*)

lemma *full_Suc_Node3_iff* [*simp*]:
full (Suc n) (Node3 l p m q r) $\longleftrightarrow$ *full n l* $\wedge$ *full n m* $\wedge$ *full n r*
by (*auto elim*: *full_elims intro*: *full.intros*)

lemma *full_Suc_Node4_iff* [*simp*]:
full (Suc n) (Node4 l p m q m′ q′ r) $\longleftrightarrow$ *full n l* $\wedge$ *full n m* $\wedge$ *full n m′* $\wedge$ *full n r*
by (*auto elim*: *full_elims intro*: *full.intros*)

lemma *full_imp_height*: *full n t* $\Longrightarrow$ *height t = n*
by (*induct set*: *full*, *simp_all*)

lemma *full_imp_bal*: *full n t* $\Longrightarrow$ *bal t*
by (*induct set*: *full*, *auto dest*: *full_imp_height*)

lemma *bal_imp_full*: *bal t* $\Longrightarrow$ *full (height t) t*
by (*induct t*, *simp_all*)

lemma *bal_iff_full*: *bal t* $\longleftrightarrow$ ($\exists$ *n*. *full n t*)
by (*auto elim*!: *bal_imp_full full_imp_bal*)

The *insert* function either preserves the height of the tree, or increases it by one. The constructor returned by the *insert* function determines which: A return value of the form T_i *t* indicates that the height will be the same. A value of the form Up_i *l p r* indicates an increase in height.

primrec $full_i$ :: *nat* $\Rightarrow$ *'a* up_i $\Rightarrow$ *bool* **where**
$full_i$ *n* (T_i *t*) $\longleftrightarrow$ *full n t* |
$full_i$ *n* (Up_i *l p r*) $\longleftrightarrow$ *full n l* $\wedge$ *full n r*

lemma $full_i$*_ins*: *full n t* $\Longrightarrow$ $full_i$ *n* (*ins a t*)
by (*induct rule*: *full.induct*) (*auto*, *auto split*: up_i*.split*)

The *insert* operation preserves balance.

lemma *bal_insert*: *bal t* $\Longrightarrow$ *bal* (*insert a t*)
unfolding *bal_iff_full insert_def*
apply (*erule exE*)
apply (*drule full*$_i$*_ins* [*of* _ _ *a*])
apply (*cases ins a t*)
apply (*auto intro*: *full.intros*)
done

22.3.2 Proofs for delete

instantiation up_d :: (*type*)*height*
begin

fun *height_up*$_d$ :: $'a$ up_d $\Rightarrow$ *nat* **where**
height (T_d *t*) = *height t* |
height (Up_d *t*) = *height t* + *1*

instance ..

end

lemma *bal_tree*$_d$*_node21*:
⟦*bal r*; *bal* (*tree*$_d$ *l*); *height r* = *height l* ⟧ $\Longrightarrow$ *bal* (*tree*$_d$ (*node21 l a r*))
by(*induct l a r rule*: *node21.induct*) *auto*

lemma *bal_tree*$_d$*_node22*:
⟦*bal*(*tree*$_d$ *r*); *bal l*; *height r* = *height l* ⟧ $\Longrightarrow$ *bal* (*tree*$_d$ (*node22 l a r*))
by(*induct l a r rule*: *node22.induct*) *auto*

lemma *bal_tree*$_d$*_node31*:
⟦ *bal* (*tree*$_d$ *l*); *bal m*; *bal r*; *height l* = *height r*; *height m* = *height r* ⟧
$\Longrightarrow$ *bal* (*tree*$_d$ (*node31 l a m b r*))
by(*induct l a m b r rule*: *node31.induct*) *auto*

lemma *bal_tree*$_d$*_node32*:
⟦ *bal l*; *bal* (*tree*$_d$ *m*); *bal r*; *height l* = *height r*; *height m* = *height r* ⟧
$\Longrightarrow$ *bal* (*tree*$_d$ (*node32 l a m b r*))
by(*induct l a m b r rule*: *node32.induct*) *auto*

lemma *bal_tree*$_d$*_node33*:
⟦ *bal l*; *bal m*; *bal*(*tree*$_d$ *r*); *height l* = *height r*; *height m* = *height r* ⟧
$\Longrightarrow$ *bal* (*tree*$_d$ (*node33 l a m b r*))
by(*induct l a m b r rule*: *node33.induct*) *auto*

lemma *bal_tree$_d$_node41*:
⟦ *bal* (*tree$_d$ l*); *bal m*; *bal n*; *bal r*; *height l* = *height r*; *height m* = *height r*; *height n* = *height r* ⟧
⟹ *bal* (*tree$_d$* (*node41 l a m b n c r*))
by(*induct l a m b n c r rule*: *node41.induct*) *auto*

lemma *bal_tree$_d$_node42*:
⟦ *bal l*; *bal* (*tree$_d$ m*); *bal n*; *bal r*; *height l* = *height r*; *height m* = *height r*; *height n* = *height r* ⟧
⟹ *bal* (*tree$_d$* (*node42 l a m b n c r*))
by(*induct l a m b n c r rule*: *node42.induct*) *auto*

lemma *bal_tree$_d$_node43*:
⟦ *bal l*; *bal m*; *bal* (*tree$_d$ n*); *bal r*; *height l* = *height r*; *height m* = *height r*; *height n* = *height r* ⟧
⟹ *bal* (*tree$_d$* (*node43 l a m b n c r*))
by(*induct l a m b n c r rule*: *node43.induct*) *auto*

lemma *bal_tree$_d$_node44*:
⟦ *bal l*; *bal m*; *bal n*; *bal* (*tree$_d$ r*); *height l* = *height r*; *height m* = *height r*; *height n* = *height r* ⟧
⟹ *bal* (*tree$_d$* (*node44 l a m b n c r*))
by(*induct l a m b n c r rule*: *node44.induct*) *auto*

lemmas *bals* = *bal_tree$_d$_node21 bal_tree$_d$_node22*
bal_tree$_d$_node31 bal_tree$_d$_node32 bal_tree$_d$_node33
bal_tree$_d$_node41 bal_tree$_d$_node42 bal_tree$_d$_node43 bal_tree$_d$_node44

lemma *height_node21*:
height r > *0* ⟹ *height*(*node21 l a r*) = *max* (*height l*) (*height r*) + *1*
by(*induct l a r rule*: *node21.induct*)(*simp_all add*: *max.assoc*)

lemma *height_node22*:
height l > *0* ⟹ *height*(*node22 l a r*) = *max* (*height l*) (*height r*) + *1*
by(*induct l a r rule*: *node22.induct*)(*simp_all add*: *max.assoc*)

lemma *height_node31*:
height m > *0* ⟹ *height*(*node31 l a m b r*) =
max (*height l*) (*max* (*height m*) (*height r*)) + *1*
by(*induct l a m b r rule*: *node31.induct*)(*simp_all add*: *max_def*)

lemma *height_node32*:
height r > *0* ⟹ *height*(*node32 l a m b r*) =

$max\ (height\ l)\ (max\ (height\ m)\ (height\ r))\ +\ 1$
by$(induct\ l\ a\ m\ b\ r\ rule{:}\ node32.induct)(simp\_all\ add{:}\ max\_def)$

lemma *height_node33*:
$height\ m > 0 \Longrightarrow height(node33\ l\ a\ m\ b\ r) =$
$max\ (height\ l)\ (max\ (height\ m)\ (height\ r))\ +\ 1$
by$(induct\ l\ a\ m\ b\ r\ rule{:}\ node33.induct)(simp\_all\ add{:}\ max\_def)$

lemma *height_node41*:
$height\ m > 0 \Longrightarrow height(node41\ l\ a\ m\ b\ n\ c\ r) =$
$max\ (height\ l)\ (max\ (height\ m)\ (max\ (height\ n)\ (height\ r)))\ +\ 1$
by$(induct\ l\ a\ m\ b\ n\ c\ r\ rule{:}\ node41.induct)(simp\_all\ add{:}\ max\_def)$

lemma *height_node42*:
$height\ l > 0 \Longrightarrow height(node42\ l\ a\ m\ b\ n\ c\ r) =$
$max\ (height\ l)\ (max\ (height\ m)\ (max\ (height\ n)\ (height\ r)))\ +\ 1$
by$(induct\ l\ a\ m\ b\ n\ c\ r\ rule{:}\ node42.induct)(simp\_all\ add{:}\ max\_def)$

lemma *height_node43*:
$height\ m > 0 \Longrightarrow height(node43\ l\ a\ m\ b\ n\ c\ r) =$
$max\ (height\ l)\ (max\ (height\ m)\ (max\ (height\ n)\ (height\ r)))\ +\ 1$
by$(induct\ l\ a\ m\ b\ n\ c\ r\ rule{:}\ node43.induct)(simp\_all\ add{:}\ max\_def)$

lemma *height_node44*:
$height\ n > 0 \Longrightarrow height(node44\ l\ a\ m\ b\ n\ c\ r) =$
$max\ (height\ l)\ (max\ (height\ m)\ (max\ (height\ n)\ (height\ r)))\ +\ 1$
by$(induct\ l\ a\ m\ b\ n\ c\ r\ rule{:}\ node44.induct)(simp\_all\ add{:}\ max\_def)$

lemmas *heights* = *height_node21 height_node22*
height_node31 height_node32 height_node33
height_node41 height_node42 height_node43 height_node44

lemma *height_split_min*:
$split\_min\ t = (x,\ t') \Longrightarrow height\ t > 0 \Longrightarrow bal\ t \Longrightarrow height\ t' = height\ t$
by$(induct\ t\ arbitrary{:}\ x\ t'\ rule{:}\ split\_min.induct)$
$(auto\ simp{:}\ heights\ split{:}\ prod.splits)$

lemma *height_del*: $bal\ t \Longrightarrow height(del\ x\ t) = height\ t$
by$(induction\ x\ t\ rule{:}\ del.induct)$
$(auto\ simp\ add{:}\ heights\ height\_split\_min\ split!{:}\ if\_split\ prod.split)$

lemma *bal_split_min*:
$[\![\ split\_min\ t = (x,\ t');\ bal\ t;\ height\ t > 0\ ]\!] \Longrightarrow bal\ (tree_d\ t')$
by$(induct\ t\ arbitrary{:}\ x\ t'\ rule{:}\ split\_min.induct)$

(*auto simp*: *heights height_split_min bals split*: *prod.splits*)

lemma *bal_tree$_d$_del*: *bal t* $\Longrightarrow$ *bal*(*tree$_d$*(*del x t*))
by(*induction x t rule*: *del.induct*)
(*auto simp*: *bals bal_split_min height_del height_split_min split*!: *if_split prod.split*)

corollary *bal_delete*: *bal t* $\Longrightarrow$ *bal*(*delete x t*)
by(*simp add*: *delete_def bal_tree$_d$_del*)

22.4 Overall Correctness

interpretation *S*: *Set_by_Ordered*
where *empty* = *empty* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = *bal*
proof (*standard*, *goal_cases*)
case *2* **thus** *?case* **by**(*simp add*: *isin_set*)
next
case *3* **thus** *?case* **by**(*simp add*: *inorder_insert*)
next
case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
case *6* **thus** *?case* **by**(*simp add*: *bal_insert*)
next
case *7* **thus** *?case* **by**(*simp add*: *bal_delete*)
qed (*simp add*: *empty_def*)+

end

23 2-3-4 Tree Implementation of Maps

theory *Tree234_Map*
imports
Tree234_Set
Map_Specs
begin

23.1 Map operations on 2-3-4 trees

fun *lookup* :: ('*a*::*linorder* $*$ '*b*) *tree234* $\Rightarrow$ '*a* $\Rightarrow$ '*b option* **where**
lookup Leaf x = *None* |
lookup (*Node2 l* (*a*,*b*) *r*) *x* = (*case cmp x a of*
LT $\Rightarrow$ *lookup l x* |

```
  GT ⇒ lookup r x |
  EQ ⇒ Some b) |
lookup (Node3 l (a1,b1) m (a2,b2) r) x = (case cmp x a1 of
  LT ⇒ lookup l x |
  EQ ⇒ Some b1 |
  GT ⇒ (case cmp x a2 of
         LT ⇒ lookup m x |
         EQ ⇒ Some b2 |
         GT ⇒ lookup r x)) |
lookup (Node4 t1 (a1,b1) t2 (a2,b2) t3 (a3,b3) t4) x = (case cmp x a2 of
  LT ⇒ (case cmp x a1 of
          LT ⇒ lookup t1 x | EQ ⇒ Some b1 | GT ⇒ lookup t2 x) |
  EQ ⇒ Some b2 |
  GT ⇒ (case cmp x a3 of
          LT ⇒ lookup t3 x | EQ ⇒ Some b3 | GT ⇒ lookup t4 x))

fun upd :: 'a::linorder ⇒ 'b ⇒ ('a*'b) tree234 ⇒ ('a*'b) up_i where
upd x y Leaf = Up_i Leaf (x,y) Leaf |
upd x y (Node2 l ab r) = (case cmp x (fst ab) of
   LT ⇒ (case upd x y l of
          T_i l' => T_i (Node2 l' ab r)
        | Up_i l1 ab' l2 => T_i (Node3 l1 ab' l2 ab r)) |
   EQ ⇒ T_i (Node2 l (x,y) r) |
   GT ⇒ (case upd x y r of
          T_i r' => T_i (Node2 l ab r')
        | Up_i r1 ab' r2 => T_i (Node3 l ab r1 ab' r2))) |
upd x y (Node3 l ab1 m ab2 r) = (case cmp x (fst ab1) of
   LT ⇒ (case upd x y l of
          T_i l' => T_i (Node3 l' ab1 m ab2 r)
        | Up_i l1 ab' l2 => Up_i (Node2 l1 ab' l2) ab1 (Node2 m ab2 r)) |
   EQ ⇒ T_i (Node3 l (x,y) m ab2 r) |
   GT ⇒ (case cmp x (fst ab2) of
          LT ⇒ (case upd x y m of
                  T_i m' => T_i (Node3 l ab1 m' ab2 r)
                | Up_i m1 ab' m2 => Up_i (Node2 l ab1 m1) ab' (Node2 m2
ab2 r)) |
          EQ ⇒ T_i (Node3 l ab1 m (x,y) r) |
          GT ⇒ (case upd x y r of
                  T_i r' => T_i (Node3 l ab1 m ab2 r')
                | Up_i r1 ab' r2 => Up_i (Node2 l ab1 m) ab2 (Node2 r1 ab'
r2)))) |
upd x y (Node4 t1 ab1 t2 ab2 t3 ab3 t4) = (case cmp x (fst ab2) of
   LT ⇒ (case cmp x (fst ab1) of
           LT ⇒ (case upd x y t1 of
```

```
                T_i t1' => T_i (Node4 t1' ab1 t2 ab2 t3 ab3 t4)
              | Up_i t11 q t12 => Up_i (Node2 t11 q t12) ab1 (Node3 t2 ab2
t3 ab3 t4)) |
        EQ ⇒ T_i (Node4 t1 (x,y) t2 ab2 t3 ab3 t4) |
        GT ⇒ (case upd x y t2 of
                T_i t2' => T_i (Node4 t1 ab1 t2' ab2 t3 ab3 t4)
              | Up_i t21 q t22 => Up_i (Node2 t1 ab1 t21) q (Node3 t22 ab2
t3 ab3 t4))) |
  EQ ⇒ T_i (Node4 t1 ab1 t2 (x,y) t3 ab3 t4) |
  GT ⇒ (case cmp x (fst ab3) of
        LT ⇒ (case upd x y t3 of
                T_i t3' ⇒ T_i (Node4 t1 ab1 t2 ab2 t3' ab3 t4)
              | Up_i t31 q t32 => Up_i (Node2 t1 ab1 t2) ab2 (Node3 t31
q t32 ab3 t4)) |
        EQ ⇒ T_i (Node4 t1 ab1 t2 ab2 t3 (x,y) t4) |
        GT ⇒ (case upd x y t4 of
                T_i t4' => T_i (Node4 t1 ab1 t2 ab2 t3 ab3 t4')
              | Up_i t41 q t42 => Up_i (Node2 t1 ab1 t2) ab2 (Node3 t3 ab3
t41 q t42))))
```

definition *update* :: *'a::linorder ⇒ 'b ⇒ ('a*'b) tree234 ⇒ ('a*'b) tree234* **where**
update x y t = tree_i(upd x y t)

fun *del* :: *'a::linorder ⇒ ('a*'b) tree234 ⇒ ('a*'b) up_d* **where**
del x Leaf = T_d Leaf |
del x (Node2 Leaf ab1 Leaf) = (if x=fst ab1 then Up_d Leaf else T_d(Node2 Leaf ab1 Leaf)) |
del x (Node3 Leaf ab1 Leaf ab2 Leaf) = T_d(if x=fst ab1 then Node2 Leaf ab2 Leaf
* else if x=fst ab2 then Node2 Leaf ab1 Leaf else Node3 Leaf ab1 Leaf ab2 Leaf) |*
del x (Node4 Leaf ab1 Leaf ab2 Leaf ab3 Leaf) =
* T_d(if x = fst ab1 then Node3 Leaf ab2 Leaf ab3 Leaf else*
*   if x = fst ab2 then Node3 Leaf ab1 Leaf ab3 Leaf else*
*   if x = fst ab3 then Node3 Leaf ab1 Leaf ab2 Leaf*
*   else Node4 Leaf ab1 Leaf ab2 Leaf ab3 Leaf) |*
del x (Node2 l ab1 r) = (case cmp x (fst ab1) of
* LT ⇒ node21 (del x l) ab1 r |*
* GT ⇒ node22 l ab1 (del x r) |*
* EQ ⇒ let (ab1',t) = split_min r in node22 l ab1' t) |*
del x (Node3 l ab1 m ab2 r) = (case cmp x (fst ab1) of
* LT ⇒ node31 (del x l) ab1 m ab2 r |*
* EQ ⇒ let (ab1',m') = split_min m in node32 l ab1' m' ab2 r |*

```
  GT ⇒ (case cmp x (fst ab2) of
         LT ⇒ node32 l ab1 (del x m) ab2 r |
         EQ ⇒ let (ab2',r') = split_min r in node33 l ab1 m ab2' r' |
         GT ⇒ node33 l ab1 m ab2 (del x r))) |
del x (Node4 t1 ab1 t2 ab2 t3 ab3 t4) = (case cmp x (fst ab2) of
  LT ⇒ (case cmp x (fst ab1) of
         LT ⇒ node41 (del x t1) ab1 t2 ab2 t3 ab3 t4 |
         EQ ⇒ let (ab',t2') = split_min t2 in node42 t1 ab' t2' ab2 t3 ab3
t4 |
         GT ⇒ node42 t1 ab1 (del x t2) ab2 t3 ab3 t4) |
  EQ ⇒ let (ab',t3') = split_min t3 in node43 t1 ab1 t2 ab' t3' ab3 t4 |
  GT ⇒ (case cmp x (fst ab3) of
        LT ⇒ node43 t1 ab1 t2 ab2 (del x t3) ab3 t4 |
         EQ ⇒ let (ab',t4') = split_min t4 in node44 t1 ab1 t2 ab2 t3 ab'
t4' |
         GT ⇒ node44 t1 ab1 t2 ab2 t3 ab3 (del x t4)))
```

definition *delete* :: *'a::linorder* ⇒ *('a*'b) tree234* ⇒ *('a*'b) tree234* **where**
delete x t = $tree_d$(*del x t*)

23.2 Functional correctness

lemma *lookup_map_of*:
sorted1(*inorder t*) ⟹ *lookup t x* = *map_of* (*inorder t*) *x*
by (*induction t*) (*auto simp*: *map_of_simps split*: *option.split*)

lemma *inorder_upd*:
sorted1(*inorder t*) ⟹ *inorder*($tree_i$(*upd a b t*)) = *upd_list a b* (*inorder t*)
by(*induction t*)
(*auto simp*: *upd_list_simps*, *auto simp*: *upd_list_simps split*: up_i.*splits*)

lemma *inorder_update*:
sorted1(*inorder t*) ⟹ *inorder*(*update a b t*) = *upd_list a b* (*inorder t*)
by(*simp add*: *update_def inorder_upd*)

lemma *inorder_del*: ⟦ *bal t* ; *sorted1*(*inorder t*) ⟧ ⟹
inorder($tree_d$ (*del x t*)) = *del_list x* (*inorder t*)
by(*induction t rule*: *del.induct*)
(*auto simp*: *del_list_simps inorder_nodes split_minD split*!: *if_splits prod.splits*)

lemma *inorder_delete*: ⟦ *bal t* ; *sorted1*(*inorder t*) ⟧ ⟹
inorder(*delete x t*) = *del_list x* (*inorder t*)

by(*simp add*: *delete_def inorder_del*)

23.3 Balancedness

lemma *bal_upd*: *bal t* $\Longrightarrow$ *bal* (*tree*$_i$(*upd x y t*)) $\wedge$ *height*(*upd x y t*) = *height t*
by (*induct t*) (*auto*, *auto split*!: *if_split up*$_i$.*split*)

lemma *bal_update*: *bal t* $\Longrightarrow$ *bal* (*update x y t*)
by (*simp add*: *update_def bal_upd*)

lemma *height_del*: *bal t* $\Longrightarrow$ *height*(*del x t*) = *height t*
by(*induction x t rule*: *del.induct*)
(*auto simp add*: *heights height_split_min split*!: *if_split prod.split*)

lemma *bal_tree*$_d$*_del*: *bal t* $\Longrightarrow$ *bal*(*tree*$_d$(*del x t*))
by(*induction x t rule*: *del.induct*)
(*auto simp*: *bals bal_split_min height_del height_split_min split*!: *if_split prod.split*)

corollary *bal_delete*: *bal t* $\Longrightarrow$ *bal*(*delete x t*)
by(*simp add*: *delete_def bal_tree*$_d$*_del*)

23.4 Overall Correctness

interpretation *M*: *Map_by_Ordered*
where *empty* = *empty* **and** *lookup* = *lookup* **and** *update* = *update* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = *bal*
proof (*standard*, *goal_cases*)
case *2* **thus** *?case* **by**(*simp add*: *lookup_map_of*)
next
case *3* **thus** *?case* **by**(*simp add*: *inorder_update*)
next
case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
case *6* **thus** *?case* **by**(*simp add*: *bal_update*)
next
case *7* **thus** *?case* **by**(*simp add*: *bal_delete*)
qed (*simp add*: *empty_def*)+

end

24 1-2 Brother Tree Implementation of Sets

theory *Brother12_Set*
imports
Cmp
Set_Specs
HOL−Number_Theory.Fib
begin

24.1 Data Type and Operations

datatype *'a bro* =
N0 |
N1 'a bro |
N2 'a bro 'a 'a bro |

L2 'a |
N3 'a bro 'a 'a bro 'a 'a bro

definition *empty* :: *'a bro* **where**
empty = *N0*

fun *inorder* :: *'a bro* ⇒ *'a list* **where**
inorder N0 = [] |
inorder (*N1 t*) = *inorder t* |
inorder (*N2 l a r*) = *inorder l* @ *a* # *inorder r* |
inorder (*L2 a*) = [*a*] |
inorder (*N3 t1 a1 t2 a2 t3*) = *inorder t1* @ *a1* # *inorder t2* @ *a2* # *inorder t3*

fun *isin* :: *'a bro* ⇒ *'a*::*linorder* ⇒ *bool* **where**
isin N0 x = *False* |
isin (*N1 t*) *x* = *isin t x* |
isin (*N2 l a r*) *x* =
(*case cmp x a of*
LT ⇒ *isin l x* |
EQ ⇒ *True* |
GT ⇒ *isin r x*)

fun *n1* :: *'a bro* ⇒ *'a bro* **where**
n1 (*L2 a*) = *N2 N0 a N0* |
n1 (*N3 t1 a1 t2 a2 t3*) = *N2* (*N2 t1 a1 t2*) *a2* (*N1 t3*) |
n1 t = *N1 t*

hide_const (**open**) *insert*

locale *insert*
begin

fun *n2* :: *'a bro* ⇒ *'a* ⇒ *'a bro* ⇒ *'a bro* **where**
n2 (L2 a1) a2 t = N3 N0 a1 N0 a2 t |
n2 (N3 t1 a1 t2 a2 t3) a3 (N1 t4) = N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4) |
n2 (N3 t1 a1 t2 a2 t3) a3 t4 = N3 (N2 t1 a1 t2) a2 (N1 t3) a3 t4 |
n2 t1 a1 (L2 a2) = N3 t1 a1 N0 a2 N0 |
n2 (N1 t1) a1 (N3 t2 a2 t3 a3 t4) = N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4) |
n2 t1 a1 (N3 t2 a2 t3 a3 t4) = N3 t1 a1 (N1 t2) a2 (N2 t3 a3 t4) |
n2 t1 a t2 = N2 t1 a t2

fun *ins* :: *'a*::*linorder* ⇒ *'a bro* ⇒ *'a bro* **where**
ins x N0 = L2 x |
ins x (N1 t) = n1 (ins x t) |
ins x (N2 l a r) =
(case cmp x a of
LT ⇒ *n2 (ins x l) a r* |
EQ ⇒ *N2 l a r* |
GT ⇒ *n2 l a (ins x r))*

fun *tree* :: *'a bro* ⇒ *'a bro* **where**
tree (L2 a) = N2 N0 a N0 |
tree (N3 t1 a1 t2 a2 t3) = N2 (N2 t1 a1 t2) a2 (N1 t3) |
tree t = t

definition *insert* :: *'a*::*linorder* ⇒ *'a bro* ⇒ *'a bro* **where**
insert x t = tree(*ins x t*)

end

locale *delete*
begin

fun *n2* :: *'a bro* ⇒ *'a* ⇒ *'a bro* ⇒ *'a bro* **where**
n2 (N1 t1) a1 (N1 t2) = N1 (N2 t1 a1 t2) |
n2 (N1 (N1 t1)) a1 (N2 (N1 t2) a2 (N2 t3 a3 t4)) =
N1 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) |
n2 (N1 (N1 t1)) a1 (N2 (N2 t2 a2 t3) a3 (N1 t4)) =
N1 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) |
n2 (N1 (N1 t1)) a1 (N2 (N2 t2 a2 t3) a3 (N2 t4 a4 t5)) =
N2 (N2 (N1 t1) a1 (N2 t2 a2 t3)) a3 (N1 (N2 t4 a4 t5)) |

```
n2 (N2 (N1 t1) a1 (N2 t2 a2 t3)) a3 (N1 (N1 t4)) =
  N1 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) |
n2 (N2 (N2 t1 a1 t2) a2 (N1 t3)) a3 (N1 (N1 t4)) =
  N1 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) |
n2 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) a5 (N1 (N1 t5)) =
  N2 (N1 (N2 t1 a1 t2)) a2 (N2 (N2 t3 a3 t4) a5 (N1 t5)) |
n2 t1 a1 t2 = N2 t1 a1 t2

fun split_min :: 'a bro ⇒ ('a × 'a bro) option where
split_min N0 = None |
split_min (N1 t) =
  (case split_min t of
     None ⇒ None |
     Some (a, t') ⇒ Some (a, N1 t')) |
split_min (N2 t1 a t2) =
  (case split_min t1 of
     None ⇒ Some (a, N1 t2) |
     Some (b, t1') ⇒ Some (b, n2 t1' a t2))

fun del :: 'a::linorder ⇒ 'a bro ⇒ 'a bro where
del _ N0          = N0 |
del x (N1 t)      = N1 (del x t) |
del x (N2 l a r) =
  (case cmp x a of
     LT ⇒ n2 (del x l) a r |
     GT ⇒ n2 l a (del x r) |
     EQ ⇒ (case split_min r of
             None ⇒ N1 l |
             Some (b, r') ⇒ n2 l b r'))

fun tree :: 'a bro ⇒ 'a bro where
tree (N1 t) = t |
tree t = t

definition delete :: 'a::linorder ⇒ 'a bro ⇒ 'a bro where
delete a t = tree (del a t)

end
```

24.2 Invariants

```
fun B :: nat ⇒ 'a bro set
and U :: nat ⇒ 'a bro set where
B 0 = {N0} |
```

$B\ (Suc\ h) = \{\ N2\ t1\ a\ t2 \mid t1\ a\ t2.$
$\quad t1 \in B\ h \cup U\ h \wedge t2 \in B\ h \vee t1 \in B\ h \wedge t2 \in B\ h \cup U\ h\} \mid$
$U\ 0 = \{\} \mid$
$U\ (Suc\ h) = N1\ `\ B\ h$

abbreviation $T\ h \equiv B\ h \cup U\ h$

fun $Bp :: nat \Rightarrow 'a\ bro\ set$ **where**
$Bp\ 0 = B\ 0 \cup L2\ `\ UNIV \mid$
$Bp\ (Suc\ 0) = B\ (Suc\ 0) \cup \{N3\ N0\ a\ N0\ b\ N0 | a\ b.\ True\} \mid$
$Bp\ (Suc(Suc\ h)) = B\ (Suc(Suc\ h)) \cup$
$\quad \{N3\ t1\ a\ t2\ b\ t3 \mid t1\ a\ t2\ b\ t3.\ t1 \in B\ (Suc\ h) \wedge t2 \in U\ (Suc\ h) \wedge t3 \in$
$B\ (Suc\ h)\}$

fun $Um :: nat \Rightarrow 'a\ bro\ set$ **where**
$Um\ 0 = \{\} \mid$
$Um\ (Suc\ h) = N1\ `\ T\ h$

24.3 Functional Correctness Proofs

24.3.1 Proofs for isin

lemma *isin_set*:
$\quad t \in T\ h \Longrightarrow sorted(inorder\ t) \Longrightarrow isin\ t\ x = (x \in set(inorder\ t))$
by(*induction h arbitrary*: *t*) (*fastforce simp*: *isin_simps split*: *if_splits*)+

24.3.2 Proofs for insertion

lemma *inorder_n1*: $inorder(n1\ t) = inorder\ t$
by(*cases t rule*: *n1.cases*) (*auto simp*: *sorted_lems*)

context *insert*
begin

lemma *inorder_n2*: $inorder(n2\ l\ a\ r) = inorder\ l\ @\ a\ \#\ inorder\ r$
by(*cases* (*l*,*a*,*r*) *rule*: *n2.cases*) (*auto simp*: *sorted_lems*)

lemma *inorder_tree*: $inorder(tree\ t) = inorder\ t$
by(*cases t*) *auto*

lemma *inorder_ins*: $t \in T\ h \Longrightarrow$
$\quad sorted(inorder\ t) \Longrightarrow inorder(ins\ a\ t) = ins\_list\ a\ (inorder\ t)$
by(*induction h arbitrary*: *t*) (*auto simp*: *ins_list_simps inorder_n1 inorder_n2*)

lemma *inorder_insert*: $t \in T\ h \Longrightarrow$

sorted(*inorder t*) $\Longrightarrow$ *inorder*(*insert a t*) = *ins_list a* (*inorder t*)
by(*simp add*: *insert_def inorder_ins inorder_tree*)

end

24.3.3 Proofs for deletion

context *delete*
begin

lemma *inorder_tree*: *inorder*(*tree t*) = *inorder t*
by(*cases t*) *auto*

lemma *inorder_n2*: *inorder*(*n2 l a r*) = *inorder l* @ *a* # *inorder r*
by(*cases* (*l*,*a*,*r*) *rule*: *n2.cases*) (*auto*)

lemma *inorder_split_min*:
t $\in$ *T h* $\Longrightarrow$ (*split_min t* = *None* $\longleftrightarrow$ *inorder t* = []) $\wedge$
(*split_min t* = *Some*(*a*,*t'*) $\longrightarrow$ *inorder t* = *a* # *inorder t'*)
by(*induction h arbitrary*: *t a t'*) (*auto simp*: *inorder_n2 split*: *option.splits*)

lemma *inorder_del*:
t $\in$ *T h* $\Longrightarrow$ *sorted*(*inorder t*) $\Longrightarrow$ *inorder*(*del x t*) = *del_list x* (*inorder t*)
by(*induction h arbitrary*: *t*) (*auto simp*: *del_list_simps inorder_n2*
inorder_split_min[*OF UnI1*] *inorder_split_min*[*OF UnI2*] *split*: *option.splits*)

lemma *inorder_delete*:
t $\in$ *T h* $\Longrightarrow$ *sorted*(*inorder t*) $\Longrightarrow$ *inorder*(*delete x t*) = *del_list x* (*inorder t*)
by(*simp add*: *delete_def inorder_del inorder_tree*)

end

24.4 Invariant Proofs

24.4.1 Proofs for insertion

lemma *n1_type*: *t* $\in$ *Bp h* $\Longrightarrow$ *n1 t* $\in$ *T* (*Suc h*)
by(*cases h rule*: *Bp.cases*) *auto*

context *insert*
begin

lemma *tree_type*: *t* $\in$ *Bp h* $\Longrightarrow$ *tree t* $\in$ *B h* $\cup$ *B* (*Suc h*)

by(*cases h rule*: *Bp.cases*) *auto*

lemma *n2_type*:
(*t1* ∈ *Bp h* ∧ *t2* ∈ *T h* ⟶ *n2 t1 a t2* ∈ *Bp* (*Suc h*)) ∧
(*t1* ∈ *T h* ∧ *t2* ∈ *Bp h* ⟶ *n2 t1 a t2* ∈ *Bp* (*Suc h*))
apply(*cases h rule*: *Bp.cases*)
apply (*auto*)[*2*]
apply(*rule conjI impI* | *erule conjE exE imageE* | *simp* | *erule disjE*)+
done

lemma *Bp_if_B*: *t* ∈ *B h* ⟹ *t* ∈ *Bp h*
by (*cases h rule*: *Bp.cases*) *simp_all*

An automatic proof:

lemma
(*t* ∈ *B h* ⟶ *ins x t* ∈ *Bp h*) ∧ (*t* ∈ *U h* ⟶ *ins x t* ∈ *T h*)
apply(*induction h arbitrary*: *t*)
apply (*simp*)
apply (*fastforce simp*: *Bp_if_B n2_type dest*: *n1_type*)
done

A detailed proof:

lemma *ins_type*:
shows *t* ∈ *B h* ⟹ *ins x t* ∈ *Bp h* **and** *t* ∈ *U h* ⟹ *ins x t* ∈ *T h*
proof(*induction h arbitrary*: *t*)
case *0*
{ case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *simp* **}**
next
case (*Suc h*)
{ case *1*
then obtain *t1 a t2* **where** [*simp*]: *t* = *N2 t1 a t2* **and**
t1: *t1* ∈ *T h* **and** *t2*: *t2* ∈ *T h* **and** *t12*: *t1* ∈ *B h* ∨ *t2* ∈ *B h*
by *auto*
have *?case* **if** *x* < *a*
proof −
have *n2* (*ins x t1*) *a t2* ∈ *Bp* (*Suc h*)
proof *cases*
assume *t1* ∈ *B h*
with *t2* **show** *?thesis* **by** (*simp add*: *Suc.IH*(*1*) *n2_type*)
next
assume *t1* ∉ *B h*
hence *1*: *t1* ∈ *U h* **and** *2*: *t2* ∈ *B h* **using** *t1 t12* **by** *auto*

```
      show ?thesis by (metis Suc.IH(2)[OF 1] Bp_if_B[OF 2] n2_type)
    qed
    with ⟨x < a⟩ show ?case by simp
  qed
  moreover
  have ?case if a < x
  proof −
    have n2 t1 a (ins x t2) ∈ Bp (Suc h)
    proof cases
      assume t2 ∈ B h
      with t1 show ?thesis by (simp add: Suc.IH(1) n2_type)
    next
      assume t2 ∉ B h
      hence 1: t1 ∈ B h and 2: t2 ∈ U h using t2 t12 by auto
      show ?thesis by (metis Bp_if_B[OF 1] Suc.IH(2)[OF 2] n2_type)
    qed
    with ⟨a < x⟩ show ?case by simp
  qed
  moreover
  have ?case if x = a
  proof −
    from 1 have t ∈ Bp (Suc h) by(rule Bp_if_B)
    thus ?case using ⟨x = a⟩ by simp
  qed
  ultimately show ?case by auto
 next
  case 2 thus ?case using Suc(1) n1_type by fastforce }
qed

lemma insert_type:
 t ∈ B h ⟹ insert x t ∈ B h ∪ B (Suc h)
unfolding insert_def by (metis ins_type(1) tree_type)

end
```

24.4.2 Proofs for deletion

```
lemma B_simps[simp]:
 N1 t ∈ B h = False
 L2 y ∈ B h = False
 (N3 t1 a1 t2 a2 t3) ∈ B h = False
 N0 ∈ B h ⟷ h = 0
by (cases h, auto)+
```

context *delete*
begin

lemma *n2_type1*:
  $[\![t1 \in Um\ h;\ t2 \in B\ h]\!] \Longrightarrow n2\ t1\ a\ t2 \in T\ (Suc\ h)$
apply(*cases h rule*: *Bp.cases*)
apply *auto*[*2*]
apply(*erule exE bexE conjE imageE* | *simp* | *erule disjE*)+
done

lemma *n2_type2*:
  $[\![t1 \in B\ h\ ;\ t2 \in Um\ h\ ]\!] \Longrightarrow n2\ t1\ a\ t2 \in T\ (Suc\ h)$
apply(*cases h rule*: *Bp.cases*)
apply *auto*[*2*]
apply(*erule exE bexE conjE imageE* | *simp* | *erule disjE*)+
done

lemma *n2_type3*:
  $[\![t1 \in T\ h\ ;\ t2 \in T\ h\ ]\!] \Longrightarrow n2\ t1\ a\ t2 \in T\ (Suc\ h)$
apply(*cases h rule*: *Bp.cases*)
apply *auto*[*2*]
apply(*erule exE bexE conjE imageE* | *simp* | *erule disjE*)+
done

lemma *split_minNoneN0*: $[\![t \in B\ h;\ split\_min\ t = None]\!] \Longrightarrow\ t = N0$
by (*cases t*) (*auto split*: *option.splits*)

lemma *split_minNoneN1* : $[\![t \in U\ h;\ split\_min\ t = None]\!] \Longrightarrow t = N1\ N0$
by (*cases h*) (*auto simp*: *split_minNoneN0 split*: *option.splits*)

lemma *split_min_type*:
  $t \in B\ h \Longrightarrow split\_min\ t = Some\ (a,\ t') \Longrightarrow t' \in T\ h$
  $t \in U\ h \Longrightarrow split\_min\ t = Some\ (a,\ t') \Longrightarrow t' \in Um\ h$
proof (*induction h arbitrary*: *t a t'*)
  case (*Suc h*)
  { case *1*
    then obtain *t1 a t2* **where** [*simp*]: $t = N2\ t1\ a\ t2$ **and**
      t12: $t1 \in T\ h\ t2 \in T\ h\ t1 \in B\ h \vee t2 \in B\ h$
      by *auto*
    show *?case*
    proof (*cases split_min t1*)
      case *None*
      show *?thesis*
      proof *cases*

```
      assume t1 ∈ B h
      with split_minNoneN0[OF this None] 1 show ?thesis by(auto)
    next
      assume t1 ∉ B h
      thus ?thesis using 1 None by (auto)
    qed
  next
    case [simp]: (Some bt')
    obtain b t1' where [simp]: bt' = (b,t1') by fastforce
    show ?thesis
    proof cases
      assume t1 ∈ B h
      from Suc.IH(1)[OF this] 1 have t1' ∈ T h by simp
      from n2_type3[OF this t12(2)] 1 show ?thesis by auto
    next
      assume t1 ∉ B h
      hence t1: t1 ∈ U h and t2: t2 ∈ B h using t12 by auto
      from Suc.IH(2)[OF t1] have t1' ∈ Um h by simp
      from n2_type1[OF this t2] 1 show ?thesis by auto
    qed
  qed
 }
 { case 2
  then obtain t1 where [simp]: t = N1 t1 and t1: t1 ∈ B h by auto
  show ?case
  proof (cases split_min t1)
    case None
    with split_minNoneN0[OF t1 None] 2 show ?thesis by(auto)
  next
    case [simp]: (Some bt')
    obtain b t1' where [simp]: bt' = (b,t1') by fastforce
    from Suc.IH(1)[OF t1] have t1' ∈ T h by simp
    thus ?thesis using 2 by auto
  qed
 }
qed auto

lemma del_type:
 t ∈ B h ⟹ del x t ∈ T h
 t ∈ U h ⟹ del x t ∈ Um h
proof (induction h arbitrary: x t)
 case (Suc h)
 { case 1
  then obtain l a r where [simp]: t = N2 l a r and
```

```
  lr: l ∈ T h r ∈ T h l ∈ B h ∨ r ∈ B h by auto
have ?case if x < a
proof cases
  assume l ∈ B h
  from n2_type3[OF Suc.IH(1)[OF this] lr(2)]
  show ?thesis using ‹x<a› by(simp)
next
  assume l ∉ B h
  hence l ∈ U h r ∈ B h using lr by auto
  from n2_type1[OF Suc.IH(2)[OF this(1)] this(2)]
  show ?thesis using ‹x<a› by(simp)
qed
moreover
have ?case if x > a
proof cases
  assume r ∈ B h
  from n2_type3[OF lr(1) Suc.IH(1)[OF this]]
  show ?thesis using ‹x>a› by(simp)
next
  assume r ∉ B h
  hence l ∈ B h r ∈ U h using lr by auto
  from n2_type2[OF this(1) Suc.IH(2)[OF this(2)]]
  show ?thesis using ‹x>a› by(simp)
qed
moreover
have ?case if [simp]: x=a
proof (cases split_min r)
  case None
  show ?thesis
  proof cases
    assume r ∈ B h
    with split_minNoneN0[OF this None] lr show ?thesis by(simp)
  next
    assume r ∉ B h
    hence r ∈ U h using lr by auto
   with split_minNoneN1[OF this None] lr(3) show ?thesis by (simp)
  qed
next
  case [simp]: (Some br')
  obtain b r' where [simp]: br' = (b,r') by fastforce
  show ?thesis
  proof cases
    assume r ∈ B h
    from split_min_type(1)[OF this] n2_type3[OF lr(1)]
```

```
      show ?thesis by simp
    next
      assume r ∉ B h
      hence l ∈ B h and r ∈ U h using lr by auto
      from split_min_type(2)[OF this(2)] n2_type2[OF this(1)]
      show ?thesis by simp
    qed
  qed
  ultimately show ?case by auto
 }
 { case 2 with Suc.IH(1) show ?case by auto }
qed auto

lemma tree_type: t ∈ T (h+1) ⟹ tree t ∈ B (h+1) ∪ B h
by(auto)

lemma delete_type: t ∈ B h ⟹ delete x t ∈ B h ∪ B(h−1)
unfolding delete_def
by (cases h) (simp, metis del_type(1) tree_type Suc_eq_plus1 diff_Suc_1)

end
```

24.5 Overall correctness

```
interpretation Set_by_Ordered
where empty = empty and isin = isin and insert = insert.insert
and delete = delete.delete and inorder = inorder and inv = λt. ∃h. t ∈
B h
proof (standard, goal_cases)
 case 2 thus ?case by(auto intro!: isin_set)
next
 case 3 thus ?case by(auto intro!: insert.inorder_insert)
next
 case 4 thus ?case by(auto intro!: delete.inorder_delete)
next
 case 6 thus ?case using insert.insert_type by blast
next
 case 7 thus ?case using delete.delete_type by blast
qed (auto simp: empty_def)
```

24.6 Height-Size Relation

By Daniel Stüwe

```
fun fib_tree :: nat ⇒ unit bro where
```

```
  fib_tree 0 = N0
| fib_tree (Suc 0) = N2 N0 () N0
| fib_tree (Suc(Suc h)) = N2 (fib_tree (h+1)) () (N1 (fib_tree h))

fun fib' :: nat ⇒ nat where
  fib' 0 = 0
| fib' (Suc 0) = 1
| fib' (Suc(Suc h)) = 1 + fib' (Suc h) + fib' h

fun size :: 'a bro ⇒ nat where
  size N0 = 0
| size (N1 t) = size t
| size (N2 t1 _ t2) = 1 + size t1 + size t2

lemma fib_tree_B: fib_tree h ∈ B h
by (induction h rule: fib_tree.induct) auto

declare [[names_short]]

lemma size_fib': size (fib_tree h) = fib' h
by (induction h rule: fib_tree.induct) auto

lemma fibfib: fib' h + 1 = fib (Suc(Suc h))
by (induction h rule: fib_tree.induct) auto

lemma B_N2_cases[consumes 1]:
assumes N2 t1 a t2 ∈ B (Suc n)
obtains
  (BB) t1 ∈ B n and t2 ∈ B n |
  (UB) t1 ∈ U n and t2 ∈ B n |
  (BU) t1 ∈ B n and t2 ∈ U n
using assms by auto

lemma size_bounded: t ∈ B h ⟹ size t ≥ size (fib_tree h)
unfolding size_fib' proof (induction h arbitrary: t rule: fib'.induct)
case (3 h t')
  note main = 3
  then obtain t1 a t2 where t': t' = N2 t1 a t2 by auto
  with main have N2 t1 a t2 ∈ B (Suc (Suc h)) by auto
  thus ?case proof (cases rule: B_N2_cases)
    case BB
    then obtain x y z where t2: t2 = N2 x y z ∨ t2 = N2 z y x x ∈ B h
by auto
     show ?thesis unfolding t' using main(1)[OF BB(1)] main(2)[OF
```

t2(2)] *t2(1)* **by** *auto*
next
case *UB*
then obtain *t11* **where** *t1*: *t1* = *N1 t11 t11* ∈ *B h* **by** *auto*
show *?thesis* **unfolding** *t' t1(1)* **using** *main(2)*[*OF t1(2)*] *main(1)*[*OF UB(2)*] **by** *simp*
next
case *BU*
then obtain *t22* **where** *t2*: *t2* = *N1 t22 t22* ∈ *B h* **by** *auto*
show *?thesis* **unfolding** *t' t2(1)* **using** *main(2)*[*OF t2(2)*] *main(1)*[*OF BU(1)*] **by** *simp*
qed
qed *auto*

theorem *t* ∈ *B h* ⟹ *fib* (*h* + *2*) ≤ *size t* + *1*
using *size_bounded*
by (*simp add*: *size_fib' fibfib*[*symmetric*] *del*: *fib.simps*)

end

25 1-2 Brother Tree Implementation of Maps

theory *Brother12_Map*
imports
Brother12_Set
Map_Specs
begin

fun *lookup* :: ('*a* × '*b*) *bro* ⇒ '*a*::*linorder* ⇒ '*b option* **where**
lookup N0 x = *None* |
lookup (*N1 t*) *x* = *lookup t x* |
lookup (*N2 l* (*a*,*b*) *r*) *x* =
(*case cmp x a of*
LT ⇒ *lookup l x* |
EQ ⇒ *Some b* |
GT ⇒ *lookup r x*)

locale *update* = *insert*
begin

fun *upd* :: '*a*::*linorder* ⇒ '*b* ⇒ ('*a*×'*b*) *bro* ⇒ ('*a*×'*b*) *bro* **where**
upd x y N0 = *L2* (*x*,*y*) |
upd x y (*N1 t*) = *n1* (*upd x y t*) |

```
upd x y (N2 l (a,b) r) =
  (case cmp x a of
     LT ⇒ n2 (upd x y l) (a,b) r |
     EQ ⇒ N2 l (a,y) r |
     GT ⇒ n2 l (a,b) (upd x y r))

definition update :: 'a::linorder ⇒ 'b ⇒ ('a×'b) bro ⇒ ('a×'b) bro where
update x y t = tree(upd x y t)

end

context delete
begin

fun del :: 'a::linorder ⇒ ('a×'b) bro ⇒ ('a×'b) bro where
del _ N0          = N0 |
del x (N1 t)      = N1 (del x t) |
del x (N2 l (a,b) r) =
  (case cmp x a of
     LT ⇒ n2 (del x l) (a,b) r |
     GT ⇒ n2 l (a,b) (del x r) |
     EQ ⇒ (case split_min r of
             None ⇒ N1 l |
             Some (ab, r') ⇒ n2 l ab r'))

definition delete :: 'a::linorder ⇒ ('a×'b) bro ⇒ ('a×'b) bro where
delete a t = tree (del a t)

end
```

25.1 Functional Correctness Proofs

25.1.1 Proofs for lookup

```
lemma lookup_map_of: t ∈ T h ⟹
  sorted1(inorder t) ⟹ lookup t x = map_of (inorder t) x
by(induction h arbitrary: t) (auto simp: map_of_simps split: option.splits)
```

25.1.2 Proofs for update

```
context update
begin

lemma inorder_upd: t ∈ T h ⟹
  sorted1(inorder t) ⟹ inorder(upd x y t) = upd_list x y (inorder t)
```

by(*induction h arbitrary*: *t*) (*auto simp*: *upd_list_simps inorder_n1 inorder_n2*)

lemma *inorder_update*: $t \in T\ h \Longrightarrow$
$sorted1(inorder\ t) \Longrightarrow inorder(update\ x\ y\ t) = upd\_list\ x\ y\ (inorder\ t)$
by(*simp add*: *update_def inorder_upd inorder_tree*)

end

25.1.3 Proofs for deletion

context *delete*
begin

lemma *inorder_del*:
$t \in T\ h \Longrightarrow sorted1(inorder\ t) \Longrightarrow inorder(del\ x\ t) = del\_list\ x\ (inorder\ t)$
by(*induction h arbitrary*: *t*) (*auto simp*: *del_list_simps inorder_n2*
inorder_split_min[*OF UnI1*] *inorder_split_min*[*OF UnI2*] *split*: *option.splits*)

lemma *inorder_delete*:
$t \in T\ h \Longrightarrow sorted1(inorder\ t) \Longrightarrow inorder(delete\ x\ t) = del\_list\ x\ (inorder\ t)$
by(*simp add*: *delete_def inorder_del inorder_tree*)

end

25.2 Invariant Proofs

25.2.1 Proofs for update

context *update*
begin

lemma *upd_type*:
$(t \in B\ h \longrightarrow upd\ x\ y\ t \in Bp\ h) \wedge (t \in U\ h \longrightarrow upd\ x\ y\ t \in T\ h)$
apply(*induction h arbitrary*: *t*)
apply (*simp*)
apply (*fastforce simp*: *Bp_if_B n2_type dest*: *n1_type*)
done

lemma *update_type*:
$t \in B\ h \Longrightarrow update\ x\ y\ t \in B\ h \cup B\ (Suc\ h)$
unfolding *update_def* **by** (*metis upd_type tree_type*)

end

25.2.2 Proofs for deletion

context *delete*
begin

lemma *del_type*:
  $t \in B\ h \Longrightarrow del\ x\ t \in T\ h$
  $t \in U\ h \Longrightarrow del\ x\ t \in Um\ h$
proof (*induction h arbitrary*: *x t*)
  case (*Suc h*)
  { case *1*
    then obtain *l a b r* **where** [*simp*]: $t = N2\ l\ (a,b)\ r$ **and**
      lr: $l \in T\ h\ r \in T\ h\ l \in B\ h \vee r \in B\ h$ **by** *auto*
    have *?case* **if** $x < a$
    proof *cases*
      assume $l \in B\ h$
      from *n2_type3*[*OF Suc.IH*(*1*)[*OF this*] *lr*(*2*)]
      show *?thesis* **using** ‹$x<a$› **by**(*simp*)
    next
      assume $l \notin B\ h$
      hence $l \in U\ h\ r \in B\ h$ **using** *lr* **by** *auto*
      from *n2_type1*[*OF Suc.IH*(*2*)[*OF this*(*1*)] *this*(*2*)]
      show *?thesis* **using** ‹$x<a$› **by**(*simp*)
    qed
    moreover
    have *?case* **if** $x > a$
    proof *cases*
      assume $r \in B\ h$
      from *n2_type3*[*OF lr*(*1*) *Suc.IH*(*1*)[*OF this*]]
      show *?thesis* **using** ‹$x>a$› **by**(*simp*)
    next
      assume $r \notin B\ h$
      hence $l \in B\ h\ r \in U\ h$ **using** *lr* **by** *auto*
      from *n2_type2*[*OF this*(*1*) *Suc.IH*(*2*)[*OF this*(*2*)]]
      show *?thesis* **using** ‹$x>a$› **by**(*simp*)
    qed
    moreover
    have *?case* **if** [*simp*]: $x=a$
    proof (*cases split_min r*)
      case *None*
      show *?thesis*
      proof *cases*
        assume $r \in B\ h$
        with *split_minNoneN0*[*OF this None*] *lr* **show** *?thesis* **by**(*simp*)

```
      next
        assume r ∉ B h
        hence r ∈ U h using lr by auto
        with split_minNoneN1[OF this None] lr(3) show ?thesis by (simp)
      qed
    next
      case [simp]: (Some br')
      obtain b r' where [simp]: br' = (b,r') by fastforce
      show ?thesis
      proof cases
        assume r ∈ B h
        from split_min_type(1)[OF this] n2_type3[OF lr(1)]
        show ?thesis by simp
      next
        assume r ∉ B h
        hence l ∈ B h and r ∈ U h using lr by auto
        from split_min_type(2)[OF this(2)] n2_type2[OF this(1)]
        show ?thesis by simp
      qed
    qed
    ultimately show ?case by auto
  }
  { case 2 with Suc.IH(1) show ?case by auto }
qed auto

lemma delete_type:
  t ∈ B h ⟹ delete x t ∈ B h ∪ B(h−1)
unfolding delete_def
by (cases h) (simp, metis del_type(1) tree_type Suc_eq_plus1 diff_Suc_1)

end
```

25.3 Overall correctness

```
interpretation Map_by_Ordered
where empty = empty and lookup = lookup and update = update.update
and delete = delete.delete and inorder = inorder and inv = λt. ∃h. t ∈
B h
proof (standard, goal_cases)
  case 2 thus ?case by(auto intro!: lookup_map_of)
next
  case 3 thus ?case by(auto intro!: update.inorder_update)
next
  case 4 thus ?case by(auto intro!: delete.inorder_delete)
```

next
case *6* **thus** *?case* **using** *update.update_type* **by** (*metis Un_iff*)
next
case *7* **thus** *?case* **using** *delete.delete_type* **by** *blast*
qed (*auto simp*: *empty_def*)

end

26 AA Tree Implementation of Sets

theory *AA_Set*
imports
Isin2
Cmp
begin

type_synonym *'a aa_tree* = (*'a*,*nat*) *tree*

definition *empty* :: *'a aa_tree* **where**
empty = *Leaf*

fun *lvl* :: *'a aa_tree* ⇒ *nat* **where**
lvl Leaf = *0* |
lvl (*Node* _ _ *lv* _) = *lv*

fun *invar* :: *'a aa_tree* ⇒ *bool* **where**
invar Leaf = *True* |
invar (*Node l a h r*) =
(*invar l* ∧ *invar r* ∧
h = *lvl l* + *1* ∧ (*h* = *lvl r* + *1* ∨ (∃ *lr b rr*. *r* = *Node lr b h rr* ∧ *h* = *lvl rr* + *1*)))

fun *skew* :: *'a aa_tree* ⇒ *'a aa_tree* **where**
skew (*Node* (*Node t1 b lvb t2*) *a lva t3*) =
(*if lva* = *lvb then Node t1 b lvb* (*Node t2 a lva t3*) *else Node* (*Node t1 b lvb t2*) *a lva t3*) |
skew t = *t*

fun *split* :: *'a aa_tree* ⇒ *'a aa_tree* **where**
split (*Node t1 a lva* (*Node t2 b lvb* (*Node t3 c lvc t4*))) =
(*if lva* = *lvb* ∧ *lvb* = *lvc* — *lva* = *lvc* suffices
then Node (*Node t1 a lva t2*) *b* (*lva*+*1*) (*Node t3 c lva t4*)
else Node t1 a lva (*Node t2 b lvb* (*Node t3 c lvc t4*))) |

```
split t = t

hide_const (open) insert

fun insert :: 'a::linorder ⇒ 'a aa_tree ⇒ 'a aa_tree where
insert x Leaf = Node Leaf x 1 Leaf |
insert x (Node t1 a lv t2) =
  (case cmp x a of
     LT ⇒ split (skew (Node (insert x t1) a lv t2)) |
     GT ⇒ split (skew (Node t1 a lv (insert x t2))) |
     EQ ⇒ Node t1 x lv t2)

fun sngl :: 'a aa_tree ⇒ bool where
sngl Leaf = False |
sngl (Node _ _ _ Leaf) = True |
sngl (Node _ _ lva (Node _ _ lvb _)) = (lva > lvb)

definition adjust :: 'a aa_tree ⇒ 'a aa_tree where
adjust t =
 (case t of
  Node l x lv r ⇒
   (if lvl l >= lv−1 ∧ lvl r >= lv−1 then t else
    if lvl r < lv−1 ∧ sngl l then skew (Node l x (lv−1) r) else
    if lvl r < lv−1
    then case l of
           Node t1 a lva (Node t2 b lvb t3)
             ⇒ Node (Node t1 a lva t2) b (lvb+1) (Node t3 x (lv−1) r)
    else
    if lvl r < lv then split (Node l x (lv−1) r)
    else
      case r of
        Node t1 b lvb t4 ⇒
          (case t1 of
             Node t2 a lva t3
               ⇒ Node (Node l x (lv−1) t2) a (lva+1)
                    (split (Node t3 b (if sngl t1 then lva else lva+1) t4)))))
```

In the paper, the last case of *adjust* is expressed with the help of an incorrect auxiliary function `nlvl`.

Function *split_max* below is called `dellrg` in the paper. The latter is incorrect for two reasons: `dellrg` is meant to delete the largest element but recurses on the left instead of the right subtree; the invariant is not restored.

```
fun split_max :: 'a aa_tree ⇒ 'a aa_tree * 'a where
split_max (Node l a lv Leaf) = (l,a) |
```

split_max (*Node l a lv r*) = (*let* (*r'*,*b*) = *split_max r in* (*adjust*(*Node l a lv r'*), *b*))

fun *delete* :: *'a*::*linorder* ⇒ *'a aa_tree* ⇒ *'a aa_tree* **where**
delete _ *Leaf* = *Leaf* |
delete x (*Node l a lv r*) =
(*case cmp x a of*
LT ⇒ *adjust* (*Node* (*delete x l*) *a lv r*) |
GT ⇒ *adjust* (*Node l a lv* (*delete x r*)) |
EQ ⇒ (*if l* = *Leaf then r*
else let (*l'*,*b*) = *split_max l in adjust* (*Node l' b lv r*)))

fun *pre_adjust* **where**
pre_adjust (*Node l a lv r*) = (*invar l* ∧ *invar r* ∧
((*lv* = *lvl l* + *1* ∧ (*lv* = *lvl r* + *1* ∨ *lv* = *lvl r* + *2* ∨ *lv* = *lvl r* ∧ *sngl r*)) ∨
(*lv* = *lvl l* + *2* ∧ (*lv* = *lvl r* + *1* ∨ *lv* = *lvl r* ∧ *sngl r*))))

declare *pre_adjust.simps* [*simp del*]

26.1 Auxiliary Proofs

lemma *split_case*: *split t* = (*case t of*
Node t1 x lvx (*Node t2 y lvy* (*Node t3 z lvz t4*)) ⇒
(*if lvx* = *lvy* ∧ *lvy* = *lvz*
then Node (*Node t1 x lvx t2*) *y* (*lvx*+*1*) (*Node t3 z lvx t4*)
else t)
| *t* ⇒ *t*)
by(*auto split*: *tree.split*)

lemma *skew_case*: *skew t* = (*case t of*
Node (*Node t1 y lvy t2*) *x lvx t3* ⇒
(*if lvx* = *lvy then Node t1 y lvx* (*Node t2 x lvx t3*) *else t*)
| *t* ⇒ *t*)
by(*auto split*: *tree.split*)

lemma *lvl_0_iff*: *invar t* ⟹ *lvl t* = *0* ⟷ *t* = *Leaf*
by(*cases t*) *auto*

lemma *lvl_Suc_iff*: *lvl t* = *Suc n* ⟷ (∃ *l a r*. *t* = *Node l a* (*Suc n*) *r*)
by(*cases t*) *auto*

lemma *lvl_skew*: *lvl* (*skew t*) = *lvl t*
by(*cases t rule*: *skew.cases*) *auto*

lemma *lvl_split*: *lvl* (*split t*) = *lvl t* ∨ *lvl* (*split t*) = *lvl t* + *1* ∧ *sngl* (*split t*)
by(*cases t rule*: *split.cases*) *auto*

lemma *invar_2Nodes*:*invar* (*Node l x lv* (*Node rl rx rlv rr*)) =
(*invar l* ∧ *invar* ⟨*rl*, *rx*, *rlv*, *rr*⟩ ∧ *lv* = *Suc* (*lvl l*) ∧
(*lv* = *Suc rlv* ∨ *rlv* = *lv* ∧ *lv* = *Suc* (*lvl rr*)))
by *simp*

lemma *invar_NodeLeaf*[*simp*]:
invar (*Node l x lv Leaf*) = (*invar l* ∧ *lv* = *Suc* (*lvl l*) ∧ *lv* = *Suc 0*)
by *simp*

lemma *sngl_if_invar*: *invar* (*Node l a n r*) ⟹ *n* = *lvl r* ⟹ *sngl r*
by(*cases r rule*: *sngl.cases*) *clarsimp*+

26.2 Invariance

26.2.1 Proofs for insert

lemma *lvl_insert_aux*:
lvl (*insert x t*) = *lvl t* ∨ *lvl* (*insert x t*) = *lvl t* + *1* ∧ *sngl* (*insert x t*)
apply(*induction t*)
apply (*auto simp*: *lvl_skew*)
apply (*metis Suc_eq_plus1 lvl.simps*(*2*) *lvl_split lvl_skew*)+
done

lemma *lvl_insert*: **obtains**
(*Same*) *lvl* (*insert x t*) = *lvl t* |
(*Incr*) *lvl* (*insert x t*) = *lvl t* + *1 sngl* (*insert x t*)
using *lvl_insert_aux* **by** *blast*

lemma *lvl_insert_sngl*: *invar t* ⟹ *sngl t* ⟹ *lvl*(*insert x t*) = *lvl t*
proof (*induction t rule*: *insert.induct*)
case (*2 x t1 a lv t2*)
consider (*LT*) *x* < *a* | (*GT*) *x* > *a* | (*EQ*) *x* = *a*
using *less_linear* **by** *blast*
thus *?case* **proof** *cases*
case *LT*
thus *?thesis* **using** *2* **by** (*auto simp add*: *skew_case split_case split*: *tree.splits*)
next
case *GT*

```
    thus ?thesis using 2 proof (cases t1)
      case Node
      thus ?thesis using 2 GT
        apply (auto simp add: skew_case split_case split: tree.splits)
            by (metis less_not_refl2 lvl.simps(2) lvl_insert_aux n_not_Suc_n
sngl.simps(3))+
    qed (auto simp add: lvl_0_iff)
  qed simp
qed simp

lemma skew_invar: invar t ⟹ skew t = t
by(cases t rule: skew.cases) auto

lemma split_invar: invar t ⟹ split t = t
by(cases t rule: split.cases) clarsimp+

lemma invar_NodeL:
  ⟦ invar(Node l x n r); invar l'; lvl l' = lvl l ⟧ ⟹ invar(Node l' x n r)
by(auto)

lemma invar_NodeR:
  ⟦ invar(Node l x n r); n = lvl r + 1; invar r'; lvl r' = lvl r ⟧ ⟹ invar(Node
l x n r')
by(auto)

lemma invar_NodeR2:
  ⟦ invar(Node l x n r); sngl r'; n = lvl r + 1; invar r'; lvl r' = n ⟧ ⟹
invar(Node l x n r')
by(cases r' rule: sngl.cases) clarsimp+


lemma lvl_insert_incr_iff: (lvl(insert a t) = lvl t + 1) ⟷
  (∃ l x r. insert a t = Node l x (lvl t + 1) r ∧ lvl l = lvl r)
apply(cases t)
apply(auto simp add: skew_case split_case split: if_splits)
apply(auto split: tree.splits if_splits)
done

lemma invar_insert: invar t ⟹ invar(insert a t)
proof(induction t)
  case N: (Node l x n r)
  hence il: invar l and ir: invar r by auto
  note iil = N.IH(1)[OF il]
  note iir = N.IH(2)[OF ir]
```

let *?t* = *Node l x n r*
have $a < x \vee a = x \vee x < a$ **by** *auto*
moreover
have *?case* **if** $a < x$
proof (*cases rule*: *lvl_insert*[*of a l*])
  case (*Same*) **thus** *?thesis*
    using ‹*a*<*x*› *invar_NodeL*[*OF N.prems iil Same*]
    by (*simp add*: *skew_invar split_invar del*: *invar.simps*)
next
  case (*Incr*)
  then obtain *t1 w t2* **where** *ial*[*simp*]: *insert a l* = *Node t1 w n t2*
    using *N.prems* **by** (*auto simp*: *lvl_Suc_iff*)
  have *l12*: *lvl t1* = *lvl t2*
    by (*metis Incr*(*1*) *ial lvl_insert_incr_iff tree.inject*)
  have *insert a ?t* = *split*(*skew*(*Node* (*insert a l*) *x n r*))
    by(*simp add*: ‹*a*<*x*›)
  also have *skew*(*Node* (*insert a l*) *x n r*) = *Node t1 w n* (*Node t2 x n r*)
    by(*simp*)
  also have *invar*(*split* ...)
  proof (*cases r*)
    case *Leaf*
    hence *l* = *Leaf* **using** *N.prems* **by**(*auto simp*: *lvl_0_iff*)
    thus *?thesis* **using** *Leaf ial* **by** *simp*
  next
    case [*simp*]: (*Node t3 y m t4*)
    show *?thesis*
    proof *cases*
      assume *m* = *n* **thus** *?thesis* **using** *N*(*3*) *iil* **by**(*auto*)
    next
      assume *m* ≠ *n* **thus** *?thesis* **using** *N*(*3*) *iil l12* **by**(*auto*)
    qed
  qed
  finally show *?thesis* **.**
qed
moreover
have *?case* **if** $x < a$
proof −
  from ‹*invar ?t*› **have** *n* = *lvl r* ∨ *n* = *lvl r* + *1* **by** *auto*
  thus *?case*
  proof
    assume *0*: *n* = *lvl r*
    have *insert a ?t* = *split*(*skew*(*Node l x n* (*insert a r*)))
      using ‹*a*>*x*› **by**(*auto*)
    also have *skew*(*Node l x n* (*insert a r*)) = *Node l x n* (*insert a r*)

```
      using N.prems by(simp add: skew_case split: tree.split)
    also have invar(split ...)
    proof −
      from lvl_insert_sngl[OF ir sngl_if_invar[OF ⟨invar ?t⟩ 0], of a]
      obtain t1 y t2 where iar: insert a r = Node t1 y n t2
        using N.prems 0 by (auto simp: lvl_Suc_iff)
      from N.prems iar 0 iir
      show ?thesis by (auto simp: split_case split: tree.splits)
    qed
    finally show ?thesis .
  next
    assume 1: n = lvl r + 1
    hence sngl ?t by(cases r) auto
    show ?thesis
    proof (cases rule: lvl_insert[of a r])
      case (Same)
     show ?thesis using ⟨x<a⟩ il ir invar_NodeR[OF N.prems 1 iir Same]
       by (auto simp add: skew_invar split_invar)
    next
      case (Incr)
      thus ?thesis using invar_NodeR2[OF ⟨invar ?t⟩ Incr(2) 1 iir] 1 ⟨x
< a⟩
        by (auto simp add: skew_invar split_invar split: if_splits)
    qed
  qed
 qed
 moreover
 have a = x ⟹ ?case using N.prems by auto
 ultimately show ?case by blast
qed simp
```

26.2.2 Proofs for delete

```
lemma invarL: ASSUMPTION(invar ⟨l, a, lv, r⟩) ⟹ invar l
by(simp add: ASSUMPTION_def)

lemma invarR: ASSUMPTION(invar ⟨lv, l, a, r⟩) ⟹ invar r
by(simp add: ASSUMPTION_def)

lemma sngl_NodeI:
  sngl (Node l a lv r) ⟹ sngl (Node l' a' lv r)
by(cases r) (simp_all)
```

declare *invarL*[*simp*] *invarR*[*simp*]

lemma *pre_cases*:
assumes *pre_adjust* (*Node l x lv r*)
obtains
(*tSngl*) *invar l* ∧ *invar r* ∧
lv = *Suc* (*lvl r*) ∧ *lvl l* = *lvl r* |
(*tDouble*) *invar l* ∧ *invar r* ∧
lv = *lvl r* ∧ *Suc* (*lvl l*) = *lvl r* ∧ *sngl r* |
(*rDown*) *invar l* ∧ *invar r* ∧
lv = *Suc* (*Suc* (*lvl r*)) ∧ *lv* = *Suc* (*lvl l*) |
(*lDown_tSngl*) *invar l* ∧ *invar r* ∧
lv = *Suc* (*lvl r*) ∧ *lv* = *Suc* (*Suc* (*lvl l*)) |
(*lDown_tDouble*) *invar l* ∧ *invar r* ∧
lv = *lvl r* ∧ *lv* = *Suc* (*Suc* (*lvl l*)) ∧ *sngl r*
using *assms* **unfolding** *pre_adjust.simps*
by *auto*

declare *invar.simps*(*2*)[*simp del*] *invar_2Nodes*[*simp add*]

lemma *invar_adjust*:
assumes *pre*: *pre_adjust* (*Node l a lv r*)
shows *invar*(*adjust* (*Node l a lv r*))
using *pre* **proof** (*cases rule*: *pre_cases*)
case (*tDouble*) **thus** *?thesis* **unfolding** *adjust_def* **by** (*cases r*) (*auto simp*: *invar.simps*(*2*))
next
case (*rDown*)
from *rDown* **obtain** *llv ll la lr* **where** *l*: *l* = *Node ll la llv lr* **by** (*cases l*) *auto*
from *rDown* **show** *?thesis* **unfolding** *adjust_def* **by** (*auto simp*: *l invar.simps*(*2*) *split*: *tree.splits*)
next
case (*lDown_tDouble*)
from *lDown_tDouble* **obtain** *rlv rr ra rl* **where** *r*: *r* = *Node rl ra rlv rr* **by** (*cases r*) *auto*
from *lDown_tDouble* **and** *r* **obtain** *rrlv rrr rra rrl* **where**
rr :*rr* = *Node rrr rra rrlv rrl* **by** (*cases rr*) *auto*
from *lDown_tDouble* **show** *?thesis* **unfolding** *adjust_def r rr*
apply (*cases rl*) **apply** (*auto simp add*: *invar.simps*(*2*) *split*!: *if_split*)
using *lDown_tDouble* **by** (*auto simp*: *split_case lvl_0_iff elim*:*lvl.elims split*: *tree.split*)
qed (*auto simp*: *split_case invar.simps*(*2*) *adjust_def split*: *tree.splits*)

lemma *lvl_adjust*:
  assumes *pre_adjust* (*Node l a lv r*)
  shows *lv* = *lvl* (*adjust*(*Node l a lv r*)) ∨ *lv* = *lvl* (*adjust*(*Node l a lv r*))
+ *1*
using *assms*(*1*) **proof**(*cases rule*: *pre_cases*)
  case *lDown_tSngl* **thus** *?thesis*
    using *lvl_split*[*of* ⟨*l*, *a*, *lvl r*, *r*⟩] **by** (*auto simp*: *adjust_def*)
next
  case *lDown_tDouble* **thus** *?thesis*
    by (*auto simp*: *adjust_def invar.simps*(*2*) *split*: *tree.split*)
qed (*auto simp*: *adjust_def split*: *tree.splits*)

lemma *sngl_adjust*: **assumes** *pre_adjust* (*Node l a lv r*)
  sngl ⟨*l*, *a*, *lv*, *r*⟩ *lv* = *lvl* (*adjust* ⟨*l*, *a*, *lv*, *r*⟩)
  shows *sngl* (*adjust* ⟨*l*, *a*, *lv*, *r*⟩)
using *assms* **proof** (*cases rule*: *pre_cases*)
  case *rDown*
  thus *?thesis* **using** *assms*(*2*,*3*) **unfolding** *adjust_def*
    by (*auto simp add*: *skew_case*) (*auto split*: *tree.split*)
qed (*auto simp*: *adjust_def skew_case split_case split*: *tree.split*)

definition *post_del t t′* ==
  invar t′ ∧
  (*lvl t′* = *lvl t* ∨ *lvl t′* + *1* = *lvl t*) ∧
  (*lvl t′* = *lvl t* ∧ *sngl t* ⟶ *sngl t′*)

lemma *pre_adj_if_postR*:
  invar⟨*lv*, *l*, *a*, *r*⟩ ⟹ *post_del r r′* ⟹ *pre_adjust* ⟨*lv*, *l*, *a*, *r′*⟩
by(*cases sngl r*)
  (*auto simp*: *pre_adjust.simps post_del_def invar.simps*(*2*) *elim*: *sngl.elims*)

lemma *pre_adj_if_postL*:
  invar⟨*l*, *a*, *lv*, *r*⟩ ⟹ *post_del l l′* ⟹ *pre_adjust* ⟨*l′*, *b*, *lv*, *r*⟩
by(*cases sngl r*)
  (*auto simp*: *pre_adjust.simps post_del_def invar.simps*(*2*) *elim*: *sngl.elims*)

lemma *post_del_adjL*:
  ⟦ *invar*⟨*l*, *a*, *lv*, *r*⟩; *pre_adjust* ⟨*l′*, *b*, *lv*, *r*⟩ ⟧
  ⟹ *post_del* ⟨*l*, *a*, *lv*, *r*⟩ (*adjust* ⟨*l′*, *b*, *lv*, *r*⟩)
unfolding *post_del_def*
by (*metis invar_adjust lvl_adjust sngl_NodeI sngl_adjust lvl.simps*(*2*))

lemma *post_del_adjR*:
assumes *invar*⟨*lv*, *l*, *a*, *r*⟩ *pre_adjust* ⟨*lv*, *l*, *a*, *r′*⟩ *post_del r r′*

shows *post_del* ⟨*lv*, *l*, *a*, *r*⟩ (*adjust* ⟨*lv*, *l*, *a*, *r'*⟩)
proof(*unfold post_del_def*, *safe del*: *disjCI*)
  let *?t* = ⟨*lv*, *l*, *a*, *r*⟩
  let *?t'* = *adjust* ⟨*lv*, *l*, *a*, *r'*⟩
  show *invar ?t'* **by**(*rule invar_adjust*[*OF assms*(*2*)])
  show *lvl ?t'* = *lvl ?t* ∨ *lvl ?t'* + *1* = *lvl ?t*
    using *lvl_adjust*[*OF assms*(*2*)] **by** *auto*
  show *sngl ?t'* **if** *as*: *lvl ?t'* = *lvl ?t sngl ?t*
  proof −
    have *s*: *sngl* ⟨*lv*, *l*, *a*, *r'*⟩
    proof(*cases r'*)
      case *Leaf* **thus** *?thesis* **by** *simp*
    next
      case *Node* **thus** *?thesis* **using** *as*(*2*) *assms*(*1*,*3*)
      by (*cases r*) (*auto simp*: *post_del_def*)
    qed
    show *?thesis* **using** *as*(*1*) *sngl_adjust*[*OF assms*(*2*) *s*] **by** *simp*
  qed
qed

declare *prod.splits*[*split*]

theorem *post_split_max*:
 ⟦ *invar t*; (*t'*, *x*) = *split_max t*; *t* ≠ *Leaf* ⟧ ⟹ *post_del t t'*
proof (*induction t arbitrary*: *t' rule*: *split_max.induct*)
  case (*2 lv l a lvr rl ra rr*)
  let *?r* = ⟨*lvr*, *rl*, *ra*, *rr*⟩
  let *?t* = ⟨*lv*, *l*, *a*, *?r*⟩
  from *2.prems*(*2*) **obtain** *r'* **where** *r'*: (*r'*, *x*) = *split_max ?r*
    and [*simp*]: *t'* = *adjust* ⟨*lv*, *l*, *a*, *r'*⟩ **by** *auto*
  from *2.IH*[*OF _ r'*] ‹*invar ?t*› **have** *post*: *post_del ?r r'* **by** *simp*
  note *preR* = *pre_adj_if_postR*[*OF* ‹*invar ?t*› *post*]
  show *?case* **by** (*simp add*: *post_del_adjR*[*OF 2.prems*(*1*) *preR post*])
qed (*auto simp*: *post_del_def*)

theorem *post_delete*: *invar t* ⟹ *post_del t* (*delete x t*)
proof (*induction t*)
  case (*Node l a lv r*)

  let *?l'* = *delete x l* **and** *?r'* = *delete x r*
  let *?t* = *Node l a lv r* **let** *?t'* = *delete x ?t*

  from *Node.prems* **have** *inv_l*: *invar l* **and** *inv_r*: *invar r* **by** (*auto*)

```
  note post_l' = Node.IH(1)[OF inv_l]
  note preL = pre_adj_if_postL[OF Node.prems post_l']

  note post_r' = Node.IH(2)[OF inv_r]
  note preR = pre_adj_if_postR[OF Node.prems post_r']

  show ?case
  proof (cases rule: linorder_cases[of x a])
    case less
    thus ?thesis using Node.prems by (simp add: post_del_adjL preL)
  next
    case greater
      thus ?thesis using Node.prems by (simp add: post_del_adjR preR
post_r')
  next
    case equal
    show ?thesis
    proof cases
      assume l = Leaf thus ?thesis using equal Node.prems
        by(auto simp: post_del_def invar.simps(2))
    next
      assume l ≠ Leaf thus ?thesis using equal
      by simp (metis Node.prems inv_l post_del_adjL post_split_max pre_adj_if_postL)
    qed
  qed
qed (simp add: post_del_def)

declare invar_2Nodes[simp del]
```

26.3 Functional Correctness

26.3.1 Proofs for insert

```
lemma inorder_split: inorder(split t) = inorder t
by(cases t rule: split.cases) (auto)

lemma inorder_skew: inorder(skew t) = inorder t
by(cases t rule: skew.cases) (auto)

lemma inorder_insert:
  sorted(inorder t) ⟹ inorder(insert x t) = ins_list x (inorder t)
by(induction t) (auto simp: ins_list_simps inorder_split inorder_skew)
```

26.3.2 Proofs for delete

lemma *inorder_adjust*: $t \neq Leaf \Longrightarrow pre\_adjust\ t \Longrightarrow inorder(adjust\ t) = inorder\ t$
by(*cases t*)
(*auto simp*: *adjust_def inorder_skew inorder_split invar.simps*(*2*) *pre_adjust.simps split*: *tree.splits*)

lemma *split_maxD*:
$[\![\ split\_max\ t = (t',x);\ t \neq Leaf;\ invar\ t\ ]\!] \Longrightarrow inorder\ t' @ [x] = inorder\ t$
by(*induction t arbitrary*: t' *rule*: *split_max.induct*)
(*auto simp*: *sorted_lems inorder_adjust pre_adj_if_postR post_split_max split*: *prod.splits*)

lemma *inorder_delete*:
$invar\ t \Longrightarrow sorted(inorder\ t) \Longrightarrow inorder(delete\ x\ t) = del\_list\ x\ (inorder\ t)$
by(*induction t*)
(*auto simp*: *del_list_simps inorder_adjust pre_adj_if_postL pre_adj_if_postR post_split_max post_delete split_maxD split*: *prod.splits*)

interpretation *S*: *Set_by_Ordered*
where *empty* = *empty* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = *invar*
proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** (*simp add*: *empty_def*)
next
case *2* **thus** *?case* **by**(*simp add*: *isin_set_inorder*)
next
case *3* **thus** *?case* **by**(*simp add*: *inorder_insert*)
next
case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
case *5* **thus** *?case* **by**(*simp add*: *empty_def*)
next
case *6* **thus** *?case* **by**(*simp add*: *invar_insert*)
next
case *7* **thus** *?case* **using** *post_delete* **by**(*auto simp*: *post_del_def*)
qed

end

27 AA Tree Implementation of Maps

theory *AA_Map*
imports
AA_Set
Lookup2
begin

fun *update* :: *'a::linorder* ⇒ *'b* ⇒ *('a*'b) aa_tree* ⇒ *('a*'b) aa_tree* **where**
update x y Leaf = Node Leaf (x,y) 1 Leaf |
update x y (Node t1 (a,b) lv t2) =
(*case cmp x a of*
LT ⇒ *split (skew (Node (update x y t1) (a,b) lv t2))* |
GT ⇒ *split (skew (Node t1 (a,b) lv (update x y t2)))* |
EQ ⇒ *Node t1 (x,y) lv t2*)

fun *delete* :: *'a::linorder* ⇒ *('a*'b) aa_tree* ⇒ *('a*'b) aa_tree* **where**
delete _ Leaf = Leaf |
delete x (Node l (a,b) lv r) =
(*case cmp x a of*
LT ⇒ *adjust (Node (delete x l) (a,b) lv r)* |
GT ⇒ *adjust (Node l (a,b) lv (delete x r))* |
EQ ⇒ (*if l = Leaf then r*
else let (l',ab') = split_max l in adjust (Node l' ab' lv r)))

27.1 Invariance

27.1.1 Proofs for insert

lemma *lvl_update_aux*:
lvl (update x y t) = lvl t ∨ lvl (update x y t) = lvl t + 1 ∧ sngl (update x y t)
apply(*induction t*)
apply (*auto simp*: *lvl_skew*)
apply (*metis Suc_eq_plus1 lvl.simps*(*2*) *lvl_split lvl_skew*)+
done

lemma *lvl_update*: **obtains**
(*Same*) *lvl (update x y t) = lvl t* |
(*Incr*) *lvl (update x y t) = lvl t + 1 sngl (update x y t)*
using *lvl_update_aux* **by** *fastforce*

declare *invar.simps*(*2*)[*simp*]

lemma *lvl_update_sngl*: *invar t* ⟹ *sngl t* ⟹ *lvl*(*update x y t*) = *lvl t*

proof (*induction t rule*: *update.induct*)
case (*2 x y t1 a b lv t2*)
consider (*LT*) $x < a$ | (*GT*) $x > a$ | (*EQ*) $x = a$
using *less_linear* **by** *blast*
thus *?case* **proof** *cases*
case *LT*
thus *?thesis* **using** *2* **by** (*auto simp add*: *skew_case split_case split*: *tree.splits*)
next
case *GT*
thus *?thesis* **using** *2* **proof** (*cases t1*)
case *Node*
thus *?thesis* **using** *2 GT*
apply (*auto simp add*: *skew_case split_case split*: *tree.splits*)
by (*metis less_not_refl2 lvl.simps*(*2*) *lvl_update_aux n_not_Suc_n sngl.simps*(*3*))+
qed (*auto simp add*: *lvl_0_iff*)
qed *simp*
qed *simp*

lemma *lvl_update_incr_iff*: (*lvl*(*update a b t*) = *lvl t* + *1*) ⟷
(∃ *l x r*. *update a b t* = *Node l x* (*lvl t* + *1*) *r* ∧ *lvl l* = *lvl r*)
apply(*cases t*)
apply(*auto simp add*: *skew_case split_case split*: *if_splits*)
apply(*auto split*: *tree.splits if_splits*)
done

lemma *invar_update*: *invar t* ⟹ *invar*(*update a b t*)
proof(*induction t*)
case *N*: (*Node l xy n r*)
hence *il*: *invar l* **and** *ir*: *invar r* **by** *auto*
note *iil* = *N.IH*(*1*)[*OF il*]
note *iir* = *N.IH*(*2*)[*OF ir*]
obtain *x y* **where** [*simp*]: *xy* = (*x*,*y*) **by** *fastforce*
let *?t* = *Node l xy n r*
have *a* < *x* ∨ *a* = *x* ∨ *x* < *a* **by** *auto*
moreover
have *?case* **if** *a* < *x*
proof (*cases rule*: *lvl_update*[*of a b l*])
case (*Same*) **thus** *?thesis*
using ‹*a*<*x*› *invar_NodeL*[*OF N.prems iil Same*]
by (*simp add*: *skew_invar split_invar del*: *invar.simps*)
next
case (*Incr*)

```
    then obtain t1 w t2 where ial[simp]: update a b l = Node t1 w n t2
      using N.prems by (auto simp: lvl_Suc_iff)
    have l12: lvl t1 = lvl t2
      by (metis Incr(1) ial lvl_update_incr_iff tree.inject)
    have update a b ?t = split(skew(Node (update a b l) xy n r))
      by(simp add: ‹a<x›)
    also have skew(Node (update a b l) xy n r) = Node t1 w n (Node t2 xy
n r)
      by(simp)
    also have invar(split ...)
    proof (cases r)
      case Leaf
      hence l = Leaf using N.prems by(auto simp: lvl_0_iff)
      thus ?thesis using Leaf ial by simp
    next
      case [simp]: (Node t3 y m t4)
      show ?thesis
      proof cases
        assume m = n thus ?thesis using N(3) iil by(auto)
      next
        assume m ≠ n thus ?thesis using N(3) iil l12 by(auto)
      qed
    qed
    finally show ?thesis .
  qed
  moreover
  have ?case if x < a
  proof −
    from ‹invar ?t› have n = lvl r ∨ n = lvl r + 1 by auto
    thus ?case
    proof
      assume 0: n = lvl r
      have update a b ?t = split(skew(Node l xy n (update a b r)))
        using ‹a>x› by(auto)
      also have skew(Node l xy n (update a b r)) = Node l xy n (update a
b r)
        using N.prems by(simp add: skew_case split: tree.split)
      also have invar(split ...)
      proof −
        from lvl_update_sngl[OF ir sngl_if_invar[OF ‹invar ?t› 0], of a b]
        obtain t1 p t2 where iar: update a b r = Node t1 p n t2
          using N.prems 0 by (auto simp: lvl_Suc_iff)
        from N.prems iar 0 iir
        show ?thesis by (auto simp: split_case split: tree.splits)
```

```
    qed
    finally show ?thesis .
  next
    assume 1: n = lvl r + 1
    hence sngl ?t by(cases r) auto
    show ?thesis
    proof (cases rule: lvl_update[of a b r])
      case (Same)
     show ?thesis using ‹x<a› il ir invar_NodeR[OF N.prems 1 iir Same]
        by (auto simp add: skew_invar split_invar)
    next
      case (Incr)
      thus ?thesis using invar_NodeR2[OF ‹invar ?t› Incr(2) 1 iir] 1 ‹x
< a›
        by (auto simp add: skew_invar split_invar split: if_splits)
    qed
  qed
 qed
 moreover
 have a = x ⟹ ?case using N.prems by auto
 ultimately show ?case by blast
qed simp
```

27.1.2 Proofs for delete

```
declare invar.simps(2)[simp del]

theorem post_delete: invar t ⟹ post_del t (delete x t)
proof (induction t)
 case (Node l ab lv r)

 obtain a b where [simp]: ab = (a,b) by fastforce

 let ?l' = delete x l and ?r' = delete x r
 let ?t = Node l ab lv r let ?t' = delete x ?t

 from Node.prems have inv_l: invar l and inv_r: invar r by (auto)

 note post_l' = Node.IH(1)[OF inv_l]
 note preL = pre_adj_if_postL[OF Node.prems post_l']

 note post_r' = Node.IH(2)[OF inv_r]
 note preR = pre_adj_if_postR[OF Node.prems post_r']
```

```
  show ?case
  proof (cases rule: linorder_cases[of x a])
    case less
    thus ?thesis using Node.prems by (simp add: post_del_adjL preL)
  next
    case greater
      thus ?thesis using Node.prems preR by (simp add: post_del_adjR
post_r')
  next
    case equal
    show ?thesis
    proof cases
      assume l = Leaf thus ?thesis using equal Node.prems
        by(auto simp: post_del_def invar.simps(2))
    next
      assume l ≠ Leaf thus ?thesis using equal Node.prems
        by simp (metis inv_l post_del_adjL post_split_max pre_adj_if_postL)
    qed
  qed
qed (simp add: post_del_def)
```

27.2 Functional Correctness Proofs

```
theorem inorder_update:
  sorted1(inorder t) ⟹ inorder(update x y t) = upd_list x y (inorder t)
by (induct t) (auto simp: upd_list_simps inorder_split inorder_skew)

theorem inorder_delete:
  ⟦invar t; sorted1(inorder t)⟧ ⟹
  inorder (delete x t) = del_list x (inorder t)
by(induction t)
  (auto simp: del_list_simps inorder_adjust pre_adj_if_postL pre_adj_if_postR
            post_split_max post_delete split_maxD split: prod.splits)

interpretation I: Map_by_Ordered
where empty = empty and lookup = lookup and update = update and
delete = delete
and inorder = inorder and inv = invar
proof (standard, goal_cases)
  case 1 show ?case by (simp add: empty_def)
next
  case 2 thus ?case by(simp add: lookup_map_of)
next
  case 3 thus ?case by(simp add: inorder_update)
```

next
case *4* **thus** *?case* **by**(*simp add*: *inorder_delete*)
next
case *5* **thus** *?case* **by**(*simp add*: *empty_def*)
next
case *6* **thus** *?case* **by**(*simp add*: *invar_update*)
next
case *7* **thus** *?case* **using** *post_delete* **by**(*auto simp*: *post_del_def*)
qed

end

28 Join-Based Implementation of Sets

theory *Set2_Join*
imports
Isin2
begin

This theory implements the set operations *insert*, *delete*, *union*, *inter*section and *diff*erence. The implementation is based on binary search trees. All operations are reduced to a single operation *join l x r* that joins two BSTs *l* and *r* and an element *x* such that $l < x < r$.

The theory is based on theory *HOL−Data_Structures.Tree2* where nodes have an additional field. This field is ignored here but it means that this theory can be instantiated with red-black trees (see theory `Set2_Join_RBT.thy`) and other balanced trees. This approach is very concrete and fixes the type of trees. Alternatively, one could assume some abstract type $'t$ of trees with suitable decomposition and recursion operators on it.

locale *Set2_Join* =
fixes *join* :: ('*a*::*linorder*,'*b*) *tree* ⇒ '*a* ⇒ ('*a*,'*b*) *tree* ⇒ ('*a*,'*b*) *tree*
fixes *inv* :: ('*a*,'*b*) *tree* ⇒ *bool*
assumes *set_join*: *set_tree* (*join l a r*) = *set_tree l* ∪ {*a*} ∪ *set_tree r*
assumes *bst_join*:
⟦ *bst l*; *bst r*; ∀ *x* ∈ *set_tree l*. *x* < *a*; ∀ *y* ∈ *set_tree r*. *a* < *y* ⟧
⟹ *bst* (*join l a r*)
assumes *inv_Leaf*: *inv* ⟨⟩
assumes *inv_join*: ⟦ *inv l*; *inv r* ⟧ ⟹ *inv* (*join l a r*)
assumes *inv_Node*: ⟦ *inv* (*Node l a h r*) ⟧ ⟹ *inv l* ∧ *inv r*
begin

declare *set_join* [*simp*]

28.1 *split_min*

fun *split_min* :: ('*a*,'*b*) *tree* ⇒ '*a* × ('*a*,'*b*) *tree* **where**
split_min (*Node l a* _ *r*) =
(*if l* = *Leaf then* (*a*,*r*) *else let* (*m*,*l'*) = *split_min l in* (*m*, *join l' a r*))

lemma *split_min_set*:
⟦ *split_min t* = (*m*,*t'*); *t* ≠ *Leaf* ⟧ ⟹ *m* ∈ *set_tree t* ∧ *set_tree t* = *Set.insert m* (*set_tree t'*)
proof(*induction t arbitrary*: *t'*)
case *Node* **thus** *?case* **by**(*auto split*: *prod.splits if_splits dest*: *inv_Node*)
next
case *Leaf* **thus** *?case* **by** *simp*
qed

lemma *split_min_bst*:
⟦ *split_min t* = (*m*,*t'*); *bst t*; *t* ≠ *Leaf* ⟧ ⟹ *bst t'* ∧ (∀ *x* ∈ *set_tree t'*. *m* < *x*)
proof(*induction t arbitrary*: *t'*)
case *Node* **thus** *?case* **by**(*fastforce simp*: *split_min_set bst_join split*: *prod.splits if_splits*)
next
case *Leaf* **thus** *?case* **by** *simp*
qed

lemma *split_min_inv*:
⟦ *split_min t* = (*m*,*t'*); *inv t*; *t* ≠ *Leaf* ⟧ ⟹ *inv t'*
proof(*induction t arbitrary*: *t'*)
case *Node* **thus** *?case* **by**(*auto simp*: *inv_join split*: *prod.splits if_splits dest*: *inv_Node*)
next
case *Leaf* **thus** *?case* **by** *simp*
qed

28.2 *join2*

definition *join2* :: ('*a*,'*b*) *tree* ⇒ ('*a*,'*b*) *tree* ⇒ ('*a*,'*b*) *tree* **where**
join2 l r = (*if r* = *Leaf then l else let* (*m*,*r'*) = *split_min r in join l m r'*)

lemma *set_join2*[*simp*]: *set_tree* (*join2 l r*) = *set_tree l* ∪ *set_tree r*
by(*simp add*: *join2_def split_min_set split*: *prod.split*)

lemma *bst_join2*: ⟦ *bst l*; *bst r*; ∀ *x* ∈ *set_tree l*. ∀ *y* ∈ *set_tree r*. *x* < *y* ⟧
⟹ *bst* (*join2 l r*)

by(*simp add*: *join2_def bst_join split_min_set split_min_bst split*: *prod.split*)

lemma *inv_join2*: ⟦ *inv l*; *inv r* ⟧ ⟹ *inv* (*join2 l r*)
by(*simp add*: *join2_def inv_join split_min_set split_min_inv split*: *prod.split*)

28.3 *split*

fun *split* :: ('*a*,'*b*)*tree* ⇒ '*a* ⇒ ('*a*,'*b*)*tree* × *bool* × ('*a*,'*b*)*tree* **where**
split Leaf k = (*Leaf*, *False*, *Leaf*) |
split (*Node l a* _ *r*) *x* =
(*if x* < *a then let* (*l1*,*b*,*l2*) = *split l x in* (*l1*, *b*, *join l2 a r*) *else*
if a < *x then let* (*r1*,*b*,*r2*) = *split r x in* (*join l a r1*, *b*, *r2*)
else (*l*, *True*, *r*))

lemma *split*: *split t x* = (*l*,*xin*,*r*) ⟹ *bst t* ⟹
set_tree l = {*a* ∈ *set_tree t*. *a* < *x*} ∧ *set_tree r* = {*a* ∈ *set_tree t*. *x* < *a*}
∧ (*xin* = (*x* ∈ *set_tree t*)) ∧ *bst l* ∧ *bst r*
proof(*induction t arbitrary*: *l xin r*)
case *Leaf* **thus** *?case* **by** *simp*
next
case *Node* **thus** *?case* **by**(*force split*!: *prod.splits if_splits intro*!: *bst_join*)
qed

lemma *split_inv*: *split t x* = (*l*,*xin*,*r*) ⟹ *inv t* ⟹ *inv l* ∧ *inv r*
proof(*induction t arbitrary*: *l xin r*)
case *Leaf* **thus** *?case* **by** *simp*
next
case *Node*
thus *?case* **by**(*force simp*: *inv_join split*!: *prod.splits if_splits dest*!: *inv_Node*)
qed

declare *split.simps*[*simp del*]

28.4 *insert*

definition *insert* :: '*a* ⇒ ('*a*,'*b*) *tree* ⇒ ('*a*,'*b*) *tree* **where**
insert x t = (*let* (*l*,_,*r*) = *split t x in join l x r*)

lemma *set_tree_insert*: *bst t* ⟹ *set_tree* (*insert x t*) = *Set.insert x* (*set_tree t*)
by(*auto simp add*: *insert_def split split*: *prod.split*)

lemma *bst_insert*: *bst t* ⟹ *bst* (*insert x t*)
by(*auto simp add*: *insert_def bst_join dest*: *split split*: *prod.split*)

lemma *inv_insert*: *inv t* $\Longrightarrow$ *inv* (*insert x t*)
by(*force simp*: *insert_def inv_join dest*: *split_inv split*: *prod.split*)

28.5 *delete*

definition *delete* :: $'a \Rightarrow ('a,'b)$ *tree* $\Rightarrow ('a,'b)$ *tree* **where**
delete x t = (*let* (*l*,_,*r*) = *split t x in join2 l r*)

lemma *set_tree_delete*: *bst t* $\Longrightarrow$ *set_tree* (*delete x t*) = *set_tree t* − {*x*}
by(*auto simp*: *delete_def split split*: *prod.split*)

lemma *bst_delete*: *bst t* $\Longrightarrow$ *bst* (*delete x t*)
by(*force simp add*: *delete_def intro*: *bst_join2 dest*: *split split*: *prod.split*)

lemma *inv_delete*: *inv t* $\Longrightarrow$ *inv* (*delete x t*)
by(*force simp*: *delete_def inv_join2 dest*: *split_inv split*: *prod.split*)

28.6 *union*

fun *union* :: $('a,'b)$*tree* $\Rightarrow ('a,'b)$*tree* $\Rightarrow ('a,'b)$*tree* **where**
union t1 t2 =
(*if t1* = *Leaf then t2 else*
if t2 = *Leaf then t1 else*
case t1 of Node l1 a _ *r1* $\Rightarrow$
let (*l2*,_ ,*r2*) = *split t2 a*;
l' = *union l1 l2*; r' = *union r1 r2*
in join l' *a* r')

declare *union.simps* [*simp del*]

lemma *set_tree_union*: *bst t2* $\Longrightarrow$ *set_tree* (*union t1 t2*) = *set_tree t1* ∪ *set_tree t2*
proof(*induction t1 t2 rule*: *union.induct*)
case (*1 t1 t2*)
then show *?case*
by (*auto simp*: *union.simps*[*of t1 t2*] *split split*: *tree.split prod.split*)
qed

lemma *bst_union*: ⟦ *bst t1*; *bst t2* ⟧ $\Longrightarrow$ *bst* (*union t1 t2*)
proof(*induction t1 t2 rule*: *union.induct*)
case (*1 t1 t2*)
thus *?case*
by(*fastforce simp*: *union.simps*[*of t1 t2*] *set_tree_union split intro*!: *bst_join*

split: *tree.split prod.split*)
qed

lemma *inv_union*: ⟦ *inv t1*; *inv t2* ⟧ ⟹ *inv* (*union t1 t2*)
proof(*induction t1 t2 rule*: *union.induct*)
case (*1 t1 t2*)
thus *?case*
by(*auto simp*:*union.simps*[*of t1 t2*] *inv_join split_inv*
split!: *tree.split prod.split dest*: *inv_Node*)
qed

28.7 *inter*

fun *inter* :: ('*a*,'*b*)*tree* ⇒ ('*a*,'*b*)*tree* ⇒ ('*a*,'*b*)*tree* **where**
inter t1 t2 =
(*if t1* = *Leaf then Leaf else*
if t2 = *Leaf then Leaf else*
case t1 of Node l1 a _ r1 ⇒
let (*l2*,*ain*,*r2*) = *split t2 a*;
l' = *inter l1 l2*; *r'* = *inter r1 r2*
in if ain then join l' a r' else join2 l' r')

declare *inter.simps* [*simp del*]

lemma *set_tree_inter*:
⟦ *bst t1*; *bst t2* ⟧ ⟹ *set_tree* (*inter t1 t2*) = *set_tree t1* ∩ *set_tree t2*
proof(*induction t1 t2 rule*: *inter.induct*)
case (*1 t1 t2*)
show *?case*
proof (*cases t1*)
case *Leaf* **thus** *?thesis* **by** (*simp add*: *inter.simps*)
next
case [*simp*]: (*Node l1 a _ r1*)
show *?thesis*
proof (*cases t2* = *Leaf*)
case *True* **thus** *?thesis* **by** (*simp add*: *inter.simps*)
next
case *False*
let *?L1* = *set_tree l1* **let** *?R1* = *set_tree r1*
have ∗: *a* ∉ *?L1* ∪ *?R1* **using** ‹*bst t1*› **by** (*fastforce*)
obtain *l2 ain r2* **where** *sp*: *split t2 a* = (*l2*,*ain*,*r2*) **using** *prod_cases3*
by *blast*
let *?L2* = *set_tree l2* **let** *?R2* = *set_tree r2* **let** *?K* = *if ain then* {*a*}

```
else {}
    have t2: set_tree t2 = ?L2 ∪ ?R2 ∪ ?K and
      **: ?L2 ∩ ?R2 = {} a ∉ ?L2 ∪ ?R2 ?L1 ∩ ?R2 = {} ?L2 ∩ ?R1
= {}
     using split[OF sp] ‹bst t1› ‹bst t2› by (force, force, force, force, force)
    have IHl: set_tree (inter l1 l2) = set_tree l1 ∩ set_tree l2
      using 1.IH(1)[OF _ False _ sp[symmetric]] 1.prems(1,2) split[OF
sp] by simp
    have IHr: set_tree (inter r1 r2) = set_tree r1 ∩ set_tree r2
      using 1.IH(2)[OF _ False _ sp[symmetric]] 1.prems(1,2) split[OF
sp] by simp
    have set_tree t1 ∩ set_tree t2 = (?L1 ∪ ?R1 ∪ {a}) ∩ (?L2 ∪ ?R2 ∪
?K)
      by(simp add: t2)
    also have ... = (?L1 ∩ ?L2) ∪ (?R1 ∩ ?R2) ∪ ?K
      using * ** by auto
    also have ... = set_tree (inter t1 t2)
    using IHl IHr sp inter.simps[of t1 t2] False by(simp)
    finally show ?thesis by simp
  qed
 qed
qed

lemma bst_inter: ⟦ bst t1; bst t2 ⟧ ⟹ bst (inter t1 t2)
proof(induction t1 t2 rule: inter.induct)
 case (1 t1 t2)
 thus ?case
  by(fastforce simp: inter.simps[of t1 t2] set_tree_inter split Let_def
     intro!: bst_join bst_join2 split: tree.split prod.split)
qed

lemma inv_inter: ⟦ inv t1; inv t2 ⟧ ⟹ inv (inter t1 t2)
proof(induction t1 t2 rule: inter.induct)
 case (1 t1 t2)
 thus ?case
  by(auto simp: inter.simps[of t1 t2] inv_join inv_join2 split_inv Let_def
     split!: tree.split prod.split dest: inv_Node)
qed
```

28.8 *diff*

```
fun diff :: ('a,'b)tree ⇒ ('a,'b)tree ⇒ ('a,'b)tree where
diff t1 t2 =
 (if t1 = Leaf then Leaf else
```

```
  if t2 = Leaf then t1 else
  case t2 of Node l2 a _ r2 ⇒
  let (l1,_,r1) = split t1 a;
      l' = diff l1 l2; r' = diff r1 r2
  in join2 l' r')

declare diff.simps [simp del]

lemma set_tree_diff:
  ⟦ bst t1; bst t2 ⟧ ⟹ set_tree (diff t1 t2) = set_tree t1 − set_tree t2
proof(induction t1 t2 rule: diff.induct)
  case (1 t1 t2)
  show ?case
  proof (cases t2)
    case Leaf thus ?thesis by (simp add: diff.simps)
  next
    case [simp]: (Node l2 a _ r2)
    show ?thesis
    proof (cases t1 = Leaf)
      case True thus ?thesis by (simp add: diff.simps)
    next
      case False
      let ?L2 = set_tree l2 let ?R2 = set_tree r2
     obtain l1 ain r1 where sp: split t1 a = (l1,ain,r1) using prod_cases3
by blast
      let ?L1 = set_tree l1 let ?R1 = set_tree r1 let ?K = if ain then {a}
else {}
      have t1: set_tree t1 = ?L1 ∪ ?R1 ∪ ?K and
           **: a ∉ ?L1 ∪ ?R1 ?L1 ∩ ?R2 = {} ?L2 ∩ ?R1 = {}
        using split[OF sp] ‹bst t1› ‹bst t2› by (force, force, force, force)
      have IHl: set_tree (diff l1 l2) = set_tree l1 − set_tree l2
        using 1.IH(1)[OF False _ _ sp[symmetric]] 1.prems(1,2) split[OF
sp] by simp
      have IHr: set_tree (diff r1 r2) = set_tree r1 − set_tree r2
        using 1.IH(2)[OF False _ _ sp[symmetric]] 1.prems(1,2) split[OF
sp] by simp
      have set_tree t1 − set_tree t2 = (?L1 ∪ ?R1) − (?L2 ∪ ?R2 ∪ {a})
        by(simp add: t1)
      also have ... = (?L1 − ?L2) ∪ (?R1 − ?R2)
        using ** by auto
      also have ... = set_tree (diff t1 t2)
      using IHl IHr sp diff.simps[of t1 t2] False by(simp)
      finally show ?thesis by simp
    qed
```

```
  qed
qed

lemma bst_diff: ⟦ bst t1; bst t2 ⟧ ⟹ bst (diff t1 t2)
proof(induction t1 t2 rule: diff.induct)
  case (1 t1 t2)
  thus ?case
    by(fastforce simp: diff.simps[of t1 t2] set_tree_diff split Let_def
        intro!: bst_join bst_join2 split: tree.split prod.split)
qed

lemma inv_diff: ⟦ inv t1; inv t2 ⟧ ⟹ inv (diff t1 t2)
proof(induction t1 t2 rule: diff.induct)
  case (1 t1 t2)
  thus ?case
    by(auto simp: diff.simps[of t1 t2] inv_join inv_join2 split_inv Let_def
        split!: tree.split prod.split dest: inv_Node)
qed
```

Locale *Set2_Join* implements locale *Set2*:

```
sublocale Set2
where empty = Leaf and insert = insert and delete = delete and isin =
isin
and union = union and inter = inter and diff = diff
and set = set_tree and invar = λt. inv t ∧ bst t
proof (standard, goal_cases)
  case 1 show ?case by (simp)
next
  case 2 thus ?case by(simp add: isin_set_tree)
next
  case 3 thus ?case by (simp add: set_tree_insert)
next
  case 4 thus ?case by (simp add: set_tree_delete)
next
  case 5 thus ?case by (simp add: inv_Leaf)
next
  case 6 thus ?case by (simp add: bst_insert inv_insert)
next
  case 7 thus ?case by (simp add: bst_delete inv_delete)
next
  case 8 thus ?case by(simp add: set_tree_union)
next
  case 9 thus ?case by(simp add: set_tree_inter)
next
```

case *10* **thus** *?case* **by**(*simp add*: *set_tree_diff*)
next
case *11* **thus** *?case* **by** (*simp add*: *bst_union inv_union*)
next
case *12* **thus** *?case* **by** (*simp add*: *bst_inter inv_inter*)
next
case *13* **thus** *?case* **by** (*simp add*: *bst_diff inv_diff*)
qed

end

interpretation *unbal*: *Set2_Join*
where *join* = $\lambda l\ x\ r$. *Node* $l\ x\ ()\ r$ **and** *inv* = λt. *True*
proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *simp*
next
case *3* **thus** *?case* **by** *simp*
next
case *4* **thus** *?case* **by** *simp*
next
case *5* **thus** *?case* **by** *simp*
qed

end

29 Join-Based Implementation of Sets via RBTs

theory *Set2_Join_RBT*
imports
Set2_Join
RBT_Set
begin

29.1 Code

Function *joinL* joins two trees (and an element). Precondition: *bheight* $l \leq$ *bheight* 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.

fun *joinL* :: $'a\ rbt \Rightarrow 'a \Rightarrow 'a\ rbt \Rightarrow 'a\ rbt$ **where**
joinL $l\ x\ r$ =
(*if bheight* l = *bheight* r *then* $R\ l\ x\ r$

```
  else case r of
    B l' x' r' ⇒ baliL (joinL l x l') x' r' |
    R l' x' r' ⇒ R (joinL l x l') x' r')

fun joinR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt where
joinR l x r =
  (if bheight l ≤ bheight r then R l x r
   else case l of
     B l' x' r' ⇒ baliR l' x' (joinR r' x r) |
     R l' x' r' ⇒ R l' x' (joinR r' x r))

fun join :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt where
join l x r =
  (if bheight l > bheight r
   then paint Black (joinR l x r)
   else if bheight l < bheight r
   then paint Black (joinL l x r)
   else B l x r)

declare joinL.simps[simp del]
declare joinR.simps[simp del]
```

One would expect *joinR* to be be completely dual to *joinL*. Thus the condition should be *bheight* l = *bheight* r. What we have done is totalize the function. On the intended domain (*bheight* $r \leq$ *bheight* l) the two versions behave exactly the same, including complexity. Thus from a programmer's perspective they are equivalent. However, not from a verifier's perspective: the total version of *joinR* is easier to reason about because lemmas about it may not require preconditions. In particular *set_tree* (*joinR* l x r) = *set_tree* $l \cup \{x\} \cup$ *set_tree* r is provable outright and hence also *set_tree* (*join* l x r) = *set_tree* $l \cup \{x\} \cup$ *set_tree* r. This is necessary because locale *Set2_Join* unconditionally assumes exactly that. Adding preconditions to this assumptions significantly complicates the proofs within *Set2_Join*, which we want to avoid.

Why not work with the partial version of *joinR* and add the precondition *bheight* $r \leq$ *bheight* l to lemmas about *joinR*? After all, that is how we worked with *joinL*, and *join* ensures that *joinL* and *joinR* are only called under the respective precondition. But function *bheight* makes the difference: it descends along the left spine, just like *joinL*. Function *joinR*, however, descends along the right spine and thus *bheight* may change all the time. Thus we would need the further precondition *invh* l. This is what we really wanted to avoid in order to satisfy the unconditional assumption in *Set2_Join*.

29.2 Properties

29.2.1 Color and height invariants

lemma *invc2_joinL*:
⟦ *invc l*; *invc r*; *bheight l* ≤ *bheight r* ⟧ ⟹
invc2 (*joinL l x r*)
∧ (*bheight l* ≠ *bheight r* ∧ *color r* = *Black* ⟶ *invc*(*joinL l x r*))
proof (*induct l x r rule*: *joinL.induct*)
case (*1 l x r*) **thus** *?case*
by(*auto simp*: *invc_baliL invc2I joinL.simps*[*of l x r*] *split*!: *tree.splits if_splits*)
qed

lemma *invc2_joinR*:
⟦ *invc l*; *invh l*; *invc r*; *invh r*; *bheight l* ≥ *bheight r* ⟧ ⟹
invc2 (*joinR l x r*)
∧ (*bheight l* ≠ *bheight r* ∧ *color l* = *Black* ⟶ *invc*(*joinR l x r*))
proof (*induct l x r rule*: *joinR.induct*)
case (*1 l x r*) **thus** *?case*
by(*fastforce simp*: *invc_baliR invc2I joinR.simps*[*of l x r*] *split*!: *tree.splits if_splits*)
qed

lemma *bheight_joinL*:
⟦ *invh l*; *invh r*; *bheight l* ≤ *bheight r* ⟧ ⟹ *bheight* (*joinL l x r*) = *bheight r*
proof (*induct l x r rule*: *joinL.induct*)
case (*1 l x r*) **thus** *?case*
by(*auto simp*: *bheight_baliL joinL.simps*[*of l x r*] *split*!: *tree.split*)
qed

lemma *invh_joinL*:
⟦ *invh l*; *invh r*; *bheight l* ≤ *bheight r* ⟧ ⟹ *invh* (*joinL l x r*)
proof (*induct l x r rule*: *joinL.induct*)
case (*1 l x r*) **thus** *?case*
by(*auto simp*: *invh_baliL bheight_joinL joinL.simps*[*of l x r*] *split*!: *tree.split color.split*)
qed

lemma *bheight_baliR*:
bheight l = *bheight r* ⟹ *bheight* (*baliR l a r*) = *Suc* (*bheight l*)
by (*cases* (*l*,*a*,*r*) *rule*: *baliR.cases*) *auto*

lemma *bheight_joinR*:
⟦ *invh l*; *invh r*; *bheight l* ≥ *bheight r* ⟧ ⟹ *bheight* (*joinR l x r*) = *bheight l*
proof (*induct l x r rule*: *joinR.induct*)
case (*1 l x r*) **thus** *?case*
by(*fastforce simp*: *bheight_baliR joinR.simps*[*of l x r*] *split*!: *tree.split*)
qed

lemma *invh_joinR*:
⟦ *invh l*; *invh r*; *bheight l* ≥ *bheight r* ⟧ ⟹ *invh* (*joinR l x r*)
proof (*induct l x r rule*: *joinR.induct*)
case (*1 l x r*) **thus** *?case*
by(*fastforce simp*: *invh_baliR bheight_joinR joinR.simps*[*of l x r*]
split!: *tree.split color.split*)
qed

lemma *rbt_join*: ⟦ *invc l*; *invh l*; *invc r*; *invh r* ⟧ ⟹ *rbt*(*join l x r*)
by(*simp add*: *invc2_joinL invc2_joinR invc_paint_Black invh_joinL invh_joinR invh_paint rbt_def*
color_paint_Black)

To make sure the the black height is not increased unnecessarily:

lemma *bheight_paint_Black*: *bheight*(*paint Black t*) ≤ *bheight t* + *1*
by(*cases t*) *auto*

lemma ⟦ *rbt l*; *rbt r* ⟧ ⟹ *bheight*(*join l x r*) ≤ *max* (*bheight l*) (*bheight r*) + *1*
using *bheight_paint_Black*[*of joinL l x r*] *bheight_paint_Black*[*of joinR l x r*]
bheight_joinL[*of l r x*] *bheight_joinR*[*of l r x*]
by(*auto simp*: *max_def rbt_def*)

29.2.2 Inorder properties

Currently unused. Instead *set_tree* and *bst* properties are proved directly.

lemma *inorder_joinL*: *bheight l* ≤ *bheight r* ⟹ *inorder*(*joinL l x r*) = *inorder l* @ *x* # *inorder r*
proof(*induction l x r rule*: *joinL.induct*)
case (*1 l x r*)
thus *?case* **by**(*auto simp*: *inorder_baliL joinL.simps*[*of l x r*] *split*!: *tree.splits color.splits*)
qed

lemma *inorder_joinR*:

inorder(*joinR l x r*) = *inorder l* @ *x* # *inorder r*
proof(*induction l x r rule*: *joinR.induct*)
case (*1 l x r*)
thus *?case* **by** (*force simp*: *inorder_baliR joinR.simps*[*of l x r*] *split*!: *tree.splits color.splits*)
qed

lemma *inorder*(*join l x r*) = *inorder l* @ *x* # *inorder r*
by(*auto simp*: *inorder_joinL inorder_joinR inorder_paint split*!: *tree.splits color.splits if_splits*
dest!: *arg_cong*[**where** *f* = *inorder*])

29.2.3 Set and bst properties

lemma *set_baliL*:
set_tree(*baliL l a r*) = *set_tree l* ∪ {*a*} ∪ *set_tree r*
by(*cases* (*l*,*a*,*r*) *rule*: *baliL.cases*) (*auto*)

lemma *set_joinL*:
bheight l ≤ *bheight r* ⟹ *set_tree* (*joinL l x r*) = *set_tree l* ∪ {*x*} ∪ *set_tree r*
proof(*induction l x r rule*: *joinL.induct*)
case (*1 l x r*)
thus *?case* **by**(*auto simp*: *set_baliL joinL.simps*[*of l x r*] *split*!: *tree.splits color.splits*)
qed

lemma *set_baliR*:
set_tree(*baliR l a r*) = *set_tree l* ∪ {*a*} ∪ *set_tree r*
by(*cases* (*l*,*a*,*r*) *rule*: *baliR.cases*) (*auto*)

lemma *set_joinR*:
set_tree (*joinR l x r*) = *set_tree l* ∪ {*x*} ∪ *set_tree r*
proof(*induction l x r rule*: *joinR.induct*)
case (*1 l x r*)
thus *?case* **by**(*force simp*: *set_baliR joinR.simps*[*of l x r*] *split*!: *tree.splits color.splits*)
qed

lemma *set_paint*: *set_tree* (*paint c t*) = *set_tree t*
by (*cases t*) *auto*

lemma *set_join*: *set_tree* (*join l x r*) = *set_tree l* ∪ {*x*} ∪ *set_tree r*
by(*simp add*: *set_joinL set_joinR set_paint*)

lemma *bst_baliL*:
⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *k*; ∀ *x*∈*set_tree r*. *k* < *x*⟧
⟹ *bst* (*baliL l k r*)
by(*cases* (*l*,*k*,*r*) *rule*: *baliL.cases*) (*auto simp*: *ball_Un*)

lemma *bst_baliR*:
⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *k*; ∀ *x*∈*set_tree r*. *k* < *x*⟧
⟹ *bst* (*baliR l k r*)
by(*cases* (*l*,*k*,*r*) *rule*: *baliR.cases*) (*auto simp*: *ball_Un*)

lemma *bst_joinL*:
⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *k*; ∀ *y*∈*set_tree r*. *k* < *y*; *bheight l* ≤ *bheight r*⟧
⟹ *bst* (*joinL l k r*)
proof(*induction l k r rule*: *joinL.induct*)
case (*1 l x r*)
thus *?case*
by(*auto simp*: *set_baliL joinL.simps*[*of l x r*] *set_joinL ball_Un intro*!: *bst_baliL*
split!: *tree.splits color.splits*)
qed

lemma *bst_joinR*:
⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *k*; ∀ *y*∈*set_tree r*. *k* < *y* ⟧
⟹ *bst* (*joinR l k r*)
proof(*induction l k r rule*: *joinR.induct*)
case (*1 l x r*)
thus *?case*
by(*auto simp*: *set_baliR joinR.simps*[*of l x r*] *set_joinR ball_Un intro*!: *bst_baliR*
split!: *tree.splits color.splits*)
qed

lemma *bst_paint*: *bst* (*paint c t*) = *bst t*
by(*cases t*) *auto*

lemma *bst_join*:
⟦*bst l*; *bst r*; ∀ *x*∈*set_tree l*. *x* < *k*; ∀ *y*∈*set_tree r*. *k* < *y* ⟧
⟹ *bst* (*join l k r*)
by(*auto simp*: *bst_paint bst_joinL bst_joinR*)

29.2.4 Interpretation of *Set2_Join* with Red-Black Tree

global_interpretation *RBT*: *Set2_Join*
where *join* = *join* **and** *inv* = $\lambda t.$ *invc t* $\wedge$ *invh t*
defines *insert_rbt* = *RBT.insert* **and** *delete_rbt* = *RBT.delete* **and** *split_rbt* = *RBT.split*
and *join2_rbt* = *RBT.join2* **and** *split_min_rbt* = *RBT.split_min*
proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** (*rule set_join*)
next
case *2* **thus** *?case* **by** (*rule bst_join*)
next
case *3* **show** *?case* **by** *simp*
next
case *4* **thus** *?case*
by (*simp add*: *invc2_joinL invc2_joinR invc_paint_Black invh_joinL invh_joinR invh_paint*)
next
case *5* **thus** *?case* **by** *simp*
qed

The invariant does not guarantee that the root node is black. This is not required to guarantee that the height is logarithmic in the size — Exercise.

end
theory *Array_Specs*
imports *Main*
begin

Array Specifications

locale *Array* =
fixes *lookup* :: $'ar \Rightarrow nat \Rightarrow 'a$
fixes *update* :: $nat \Rightarrow 'a \Rightarrow 'ar \Rightarrow 'ar$
fixes *len* :: $'ar \Rightarrow nat$
fixes *array* :: $'a\ list \Rightarrow 'ar$

fixes *list* :: $'ar \Rightarrow 'a\ list$
fixes *invar* :: $'ar \Rightarrow bool$

assumes *lookup*: *invar ar* $\Longrightarrow$ *n* $<$ *len ar* $\Longrightarrow$ *lookup ar n* = *list ar* ! *n*
assumes *update*: *invar ar* $\Longrightarrow$ *n* $<$ *len ar* $\Longrightarrow$ *list*(*update n x ar*) = (*list ar*)[*n*:=*x*]
assumes *len_array*: *invar ar* $\Longrightarrow$ *len ar* = *length* (*list ar*)
assumes *array*: *list* (*array xs*) = *xs*

assumes *invar_update*: *invar ar* $\Longrightarrow$ *n* $<$ *len ar* $\Longrightarrow$ *invar*(*update n x ar*)

assumes *invar_array*: *invar*(*array xs*)

locale *Array_Flex* = *Array* +
fixes *add_lo* :: *'a* ⇒ *'ar* ⇒ *'ar*
fixes *del_lo* :: *'ar* ⇒ *'ar*
fixes *add_hi* :: *'a* ⇒ *'ar* ⇒ *'ar*
fixes *del_hi* :: *'ar* ⇒ *'ar*

assumes *add_lo*: *invar ar* ⟹ *list*(*add_lo a ar*) = *a* # *list ar*
assumes *del_lo*: *invar ar* ⟹ *list*(*del_lo ar*) = *tl* (*list ar*)
assumes *add_hi*: *invar ar* ⟹ *list*(*add_hi a ar*) = *list ar* @ [*a*]
assumes *del_hi*: *invar ar* ⟹ *list*(*del_hi ar*) = *butlast* (*list ar*)

assumes *invar_add_lo*: *invar ar* ⟹ *invar* (*add_lo a ar*)
assumes *invar_del_lo*: *invar ar* ⟹ *invar* (*del_lo ar*)
assumes *invar_add_hi*: *invar ar* ⟹ *invar* (*add_hi a ar*)
assumes *invar_del_hi*: *invar ar* ⟹ *invar* (*del_hi ar*)

end

30 Braun Trees

theory *Braun_Tree*
imports *HOL−Library.Tree_Real*
begin

Braun Trees were studied by Braun and Rem [5] and later Hoogerwoord [9].

fun *braun* :: *'a tree* ⇒ *bool* **where**
braun Leaf = *True* |
braun (*Node l x r*) = ((*size l* = *size r* ∨ *size l* = *size r* + *1*) ∧ *braun l* ∧ *braun r*)

lemma *braun_Node'*:
braun (*Node l x r*) = (*size r* ≤ *size l* ∧ *size l* ≤ *size r* + *1* ∧ *braun l* ∧ *braun r*)
by *auto*

The shape of a Braun-tree is uniquely determined by its size:

lemma *braun_unique*: ⟦ *braun* (*t1*::*unit tree*); *braun t2*; *size t1* = *size t2* ⟧ ⟹ *t1* = *t2*
proof (*induction t1 arbitrary*: *t2*)
case *Leaf* **thus** *?case* **by** *simp*
next

case (*Node l1 _ r1*)
from *Node.prems*(*3*) **have** *t2* ≠ *Leaf* **by** *auto*
then obtain *l2 x2 r2* **where** [*simp*]: *t2* = *Node l2 x2 r2* **by** (*meson neq_Leaf_iff*)
with *Node.prems* **have** *size l1* = *size l2* ∧ *size r1* = *size r2* **by** *auto*
thus *?case* **using** *Node.prems*(*1*,*2*) *Node.IH* **by** *auto*
qed

Braun trees are balanced:

lemma *balanced_if_braun*: *braun t* ⟹ *balanced t*
proof(*induction t*)
case *Leaf* **show** *?case* **by** (*simp add*: *balanced_def*)
next
case (*Node l x r*)
have *size l* = *size r* ∨ *size l* = *size r* + *1* (**is** *?A* ∨ *?B*)
using *Node.prems* **by** *simp*
thus *?case*
proof
assume *?A*
thus *?thesis* **using** *Node*
apply(*simp add*: *balanced_def min_def max_def*)
by (*metis Node.IH balanced_optimal le_antisym le_refl*)
next
assume *?B*
thus *?thesis* **using** *Node* **by**(*intro balanced_Node_if_wbal1*) *auto*
qed
qed

30.1 Numbering Nodes

We show that a tree is a Braun tree iff a parity-based numbering (*braun_indices*) of nodes yields an interval of numbers.

fun *braun_indices* :: *'a tree* ⇒ *nat set* **where**
braun_indices Leaf = {} |
braun_indices (*Node l _ r*) = {*1*} ∪ (∗) *2* ‘ *braun_indices l* ∪ *Suc* ‘ (∗) *2* ‘ *braun_indices r*

lemma *braun_indices1*: *0* ∉ *braun_indices t*
by (*induction t*) *auto*

lemma *finite_braun_indices*: *finite*(*braun_indices t*)
by (*induction t*) *auto*

lemma *braun_indices_if_braun*: *braun t* ⟹ *braun_indices t* = {*1*..*size t*}

proof(*induction t*)
case *Leaf* **thus** *?case* **by** *simp*
next
have ∗: (∗) *2* ‘ {*a..b*} ∪ *Suc* ‘ (∗) *2* ‘ {*a..b*} = {*2∗a..2∗b+1*} (**is** *?l* = *?r*) **for** *a b*
proof
show *?l* ⊆ *?r* **by** *auto*
next
have ∃ *x2*∈{*a..b*}. *x* ∈ {*Suc* (*2∗x2*), *2∗x2*} **if** ∗: *x* ∈ {*2∗a .. 2∗b+1*} **for** *x*
proof −
have *x div 2* ∈ {*a..b*} **using** ∗ **by** *auto*
moreover have *x* ∈ {*2* ∗ (*x div 2*), *Suc*(*2* ∗ (*x div 2*))} **by** *auto*
ultimately show *?thesis* **by** *blast*
qed
thus *?r* ⊆ *?l* **by** *fastforce*
qed
case (*Node l x r*)
hence *size l* = *size r* ∨ *size l* = *size r* + *1* (**is** *?A* ∨ *?B*) **by** *auto*
thus *?case*
proof
assume *?A*
with *Node* **show** *?thesis* **by** (*auto simp*: ∗)
next
assume *?B*
with *Node* **show** *?thesis* **by** (*auto simp*: ∗ *atLeastAtMostSuc_conv*)
qed
qed

lemma *disj_evens_odds*: (∗) *2* ‘ *A* ∩ *Suc* ‘ (∗) *2* ‘ *B* = {}
using *double_not_eq_Suc_double* **by** *auto*

lemma *Suc0_notin_double*: *Suc 0* ∉ (∗) *2* ‘ *A*
by(*auto*)

lemma *zero_in_double_iff*: (*0*::*nat*) ∈ (∗) *2* ‘ *A* ⟷ *0* ∈ *A*
by(*auto*)

lemma *Suc_in_Suc_image_iff*: *Suc n* ∈ *Suc* ‘ *A* ⟷ *n* ∈ *A*
by(*auto*)

lemmas *nat_in_image* = *Suc0_notin_double zero_in_double_iff Suc_in_Suc_image_iff*

lemma *card_braun_indices*: *card* (*braun_indices t*) = *size t*

proof (*induction t*)
case *Leaf* **thus** *?case* **by** *simp*
next
case *Node*
thus *?case*
by(*auto simp*: *UNION_singleton_eq_range finite_braun_indices card_Un_disjoint card_insert_if disj_evens_odds card_image inj_on_def braun_indices1*)
qed

lemma *disj_union_eq_iff*:
⟦ *L1* ∩ *R2* = {}; *L2* ∩ *R1* = {} ⟧ ⟹ *L1* ∪ *R1* = *L2* ∪ *R2* ⟷ *L1* = *L2* ∧ *R1* = *R2*
by *blast*

lemma *inj_braun_indices*: *braun_indices t1* = *braun_indices t2* ⟹ *t1* = (*t2*::*unit tree*)
proof(*induction t1 arbitrary*: *t2*)
case *Leaf* **thus** *?case* **using** *braun_indices.elims* **by** *blast*
next
case (*Node l1 x1 r1*)
have *t2* ≠ *Leaf*
proof
assume *t2* = *Leaf*
with *Node.prems* **show** *False* **by** *simp*
qed
thus *?case* **using** *Node*
by (*auto simp*: *neq_Leaf_iff insert_ident nat_in_image braun_indices1 disj_union_eq_iff disj_evens_odds inj_image_eq_iff inj_def*)
qed

How many even/odd natural numbers are there between m and n?

lemma *card_Icc_even_nat*:
card {*i* ∈ {*m*..*n*::*nat*}. *even i*} = (*n*+1−*m* + (*m*+1) *mod* 2) *div* 2 (**is** *?l m n* = *?r m n*)
proof(*induction n*+1 − *m arbitrary*: *n m*)
case *0* **thus** *?case* **by** *simp*
next
case *Suc*
have *m* ≤ *n* **using** *Suc*(*2*) **by** *arith*
hence {*m*..*n*} = *insert m* {*m*+1..*n*} **by** *auto*
hence *?l m n* = *card* {*i* ∈ *insert m* {*m*+1..*n*}. *even i*} **by** *simp*
also have ... = *?r m n* (**is** *?l* = *?r*)
proof (*cases*)
assume *even m*

hence $\{i \in \mathit{insert}\ m\ \{m{+}1..n\}.\ \mathit{even}\ i\} = \mathit{insert}\ m\ \{i \in \{m{+}1..n\}.\ \mathit{even}\ i\}$ **by** *auto*
hence $?l = \mathit{card}\ \{i \in \{m{+}1..n\}.\ \mathit{even}\ i\} + 1$ **by** *simp*
also have $\ldots = (n{-}m + (m{+}2)\ \mathit{mod}\ 2)\ \mathit{div}\ 2 + 1$ **using** *Suc(1)[of n m+1] Suc(2)* **by** *simp*
also have $\ldots = ?r$ **using** ‹*even m*› ‹$m \le n$› **by** *auto*
finally show *?thesis* **.**
next
assume *odd m*
hence $\{i \in \mathit{insert}\ m\ \{m{+}1..n\}.\ \mathit{even}\ i\} = \{i \in \{m{+}1..n\}.\ \mathit{even}\ i\}$ **by** *auto*
hence *?l = card ...* **by** *simp*
also have $\ldots = (n{-}m + (m{+}2)\ \mathit{mod}\ 2)\ \mathit{div}\ 2$ **using** *Suc(1)[of n m+1] Suc(2)* **by** *simp*
also have $\ldots = ?r$ **using** ‹*odd m*› ‹$m \le n$› *even_iff_mod_2_eq_zero[of m]* **by** *simp*
finally show *?thesis* **.**
qed
finally show *?case* **.**
qed

lemma *card_Icc_odd_nat*: $\mathit{card}\ \{i \in \{m..n::\mathit{nat}\}.\ \mathit{odd}\ i\} = (n{+}1{-}m + m\ \mathit{mod}\ 2)\ \mathit{div}\ 2$
proof −
let $?A = \{i \in \{m..n\}.\ \mathit{odd}\ i\}$
let $?B = \{i \in \{m{+}1..n{+}1\}.\ \mathit{even}\ i\}$
have $\mathit{card}\ ?A = \mathit{card}\ (\mathit{Suc}\ `\ ?A)$ **by** (*simp add*: *card_image*)
also have $\mathit{Suc}\ `\ ?A = ?B$ **using** *Suc_le_D* **by**(*force simp*: *image_iff*)
also have $\mathit{card}\ ?B = (n{+}1{-}m + (m)\ \mathit{mod}\ 2)\ \mathit{div}\ 2$
using *card_Icc_even_nat[of m+1 n+1]* **by** *simp*
finally show *?thesis* **.**
qed

lemma *compact_Icc_even*: **assumes** $A = \{i \in \{m..n\}.\ \mathit{even}\ i\}$
shows $A = (\lambda j.\ 2{*}(j{-}1) + m + m\ \mathit{mod}\ 2)\ `\ \{1..\mathit{card}\ A\}$ (**is** $\_ = ?A$)
proof
let $?a = (n{+}1{-}m + (m{+}1)\ \mathit{mod}\ 2)\ \mathit{div}\ 2$
have $\exists j \in \{1..?a\}.\ i = 2{*}(j{-}1) + m + m\ \mathit{mod}\ 2$ **if** $*$: $i \in \{m..n\}$ *even i* **for** *i*
proof −
let $?j = (i - (m + m\ \mathit{mod}\ 2))\ \mathit{div}\ 2 + 1$
have $?j \in \{1..?a\} \wedge i = 2{*}(?j{-}1) + m + m\ \mathit{mod}\ 2$ **using** $*$ **by**(*auto simp*: *mod2_eq_if*) *presburger+*
thus *?thesis* **by** *blast*

```
  qed
  thus A ⊆ ?A using assms
    by(auto simp: image_iff card_Icc_even_nat simp del: atLeastAtMost_iff)
next
  let ?a = (n+1−m + (m+1) mod 2) div 2
  have 1: 2 * (j − 1) + m + m mod 2 ∈ {m..n} if *: j ∈ {1..?a} for j
    using * by(auto simp: mod2_eq_if)
  have 2: even (2 * (j − 1) + m + m mod 2) for j by presburger
  show ?A ⊆ A
   apply(simp add: assms card_Icc_even_nat del: atLeastAtMost_iff One_nat_def)
    using 1 2 by blast
qed

lemma compact_Icc_odd:
  assumes B = {i ∈ {m..n}. odd i} shows B = (λi. 2*(i−1) + m +
(m+1) mod 2) ‘ {1..card B}
proof −
  define A :: nat set where A = Suc ‘ B
  have A = {i ∈ {m+1..n+1}. even i}
    using Suc_le_D by(force simp add: A_def assms image_iff)
  from compact_Icc_even[OF this]
  have A = Suc ‘ (λi. 2 * (i − 1) + m + (m + 1) mod 2) ‘ {1..card A}
    by (simp add: image_comp o_def)
  hence B: B = (λi. 2 * (i − 1) + m + (m + 1) mod 2) ‘ {1..card A}
    using A_def by (simp add: inj_image_eq_iff)
  have card A = card B by (metis A_def bij_betw_Suc bij_betw_same_card)
  with B show ?thesis by simp
qed

lemma even_odd_decomp: assumes ∀ x ∈ A. even x ∀ x ∈ B. odd x  A ∪
B = {m..n}
shows (let a = card A; b = card B in
   a + b = n+1−m ∧
   A = (λi. 2*(i−1) + m + m mod 2) ‘ {1..a} ∧
   B = (λi. 2*(i−1) + m + (m+1) mod 2) ‘ {1..b} ∧
   (a = b ∨ a = b+1 ∧ even m ∨ a+1 = b ∧ odd m))
proof −
  let ?a = card A let ?b = card B
  have finite A ∧ finite B
    by (metis ‹A ∪ B = {m..n}› finite_Un finite_atLeastAtMost)
  hence ab: ?a + ?b = Suc n − m
    by (metis Int_emptyI assms card_Un_disjoint card_atLeastAtMost)
  have A: A = {i ∈ {m..n}. even i} using assms by auto
  hence A': A = (λi. 2*(i−1) + m + m mod 2) ‘ {1..?a} by(rule com-
```

pact_Icc_even)
have *B*: $B = \{i \in \{m..n\}.\ odd\ i\}$ **using** *assms* **by** *auto*
hence *B'*: $B = (\lambda i.\ 2*(i-1) + m + (m+1)\ mod\ 2)\ `\ \{1..?b\}$ **by**(*rule compact_Icc_odd*)
have $?a = ?b \vee ?a = ?b+1 \wedge even\ m \vee ?a+1 = ?b \wedge odd\ m$
apply(*simp add*: *Let_def mod2_eq_if*
card_Icc_even_nat[*of m n*, *simplified A*[*symmetric*]]
card_Icc_odd_nat[*of m n*, *simplified B*[*symmetric*]] *split*!: *if_splits*)
by *linarith*
with *ab A' B'* **show** *?thesis* **by** *simp*
qed

lemma *braun_if_braun_indices*: $braun\_indices\ t = \{1..size\ t\} \Longrightarrow braun\ t$
proof(*induction t*)
case *Leaf*
then show *?case* **by** *simp*
next
case (*Node t1 x2 t2*)
have *1*: $i > 0 \Longrightarrow Suc(Suc(2 * (i - Suc\ 0))) = 2*i$ **for** *i*::*nat* **by**(*simp add*: *algebra_simps*)
have *2*: $i > 0 \Longrightarrow 2 * (i - Suc\ 0) + 3 = 2*i + 1$ **for** *i*::*nat* **by**(*simp add*: *algebra_simps*)
have *3*: $(*)\ 2\ `\ braun\_indices\ t1 \cup Suc\ `\ (*)\ 2\ `\ braun\_indices\ t2 = \{2..size\ t1 + size\ t2 + 1\}$ **using** *Node.prems*
by (*simp add*: *insert_ident Icc_eq_insert_lb_nat nat_in_image braun_indices1*)
thus *?case* **using** *Node.IH even_odd_decomp*[*OF _ _ 3*]
by(*simp add*: *card_image inj_on_def card_braun_indices Let_def 1 2 inj_image_eq_iff image_comp*
cong: *image_cong_simp*)
qed

lemma *braun_iff_braun_indices*: $braun\ t \longleftrightarrow braun\_indices\ t = \{1..size\ t\}$
using *braun_if_braun_indices braun_indices_if_braun* **by** *blast*

end

31 Arrays via Braun Trees

theory *Array_Braun*
imports
Array_Specs
Braun_Tree
begin

31.1 Array

fun *lookup1* :: *'a tree ⇒ nat ⇒ 'a* **where**
lookup1 (Node l x r) n = (if n=1 then x else lookup1 (if even n then l else r) (n div 2))

fun *update1* :: *nat ⇒ 'a ⇒ 'a tree ⇒ 'a tree* **where**
update1 n x Leaf = Node Leaf x Leaf |
update1 n x (Node l a r) =
(if n=1 then Node l x r else
if even n then Node (update1 (n div 2) x l) a r
else Node l a (update1 (n div 2) x r))

fun *adds* :: *'a list ⇒ nat ⇒ 'a tree ⇒ 'a tree* **where**
adds [] n t = t |
adds (x#xs) n t = adds xs (n+1) (update1 (n+1) x t)

fun *list* :: *'a tree ⇒ 'a list* **where**
list Leaf = [] |
list (Node l x r) = x # splice (list l) (list r)

31.1.1 Functional Correctness

lemma *size_list*: *size(list t) = size t*
by(*induction t*)(*auto*)

lemma *minus1_div2*: *(n − Suc 0) div 2 = (if odd n then n div 2 else n div 2 − 1)*
by *auto arith*

lemma *nth_splice*: *⟦ n < size xs + size ys; size ys ≤ size xs; size xs ≤ size ys + 1 ⟧*
⟹ splice xs ys ! n = (if even n then xs else ys) ! (n div 2)
apply(*induction xs ys arbitrary*: *n rule*: *splice.induct*)
apply (*auto simp*: *nth_Cons' minus1_div2*)
done

lemma *div2_in_bounds*:
⟦ braun (Node l x r); n ∈ {1..size(Node l x r)}; n > 1 ⟧ ⟹
(odd n ⟶ n div 2 ∈ {1..size r}) ∧ (even n ⟶ n div 2 ∈ {1..size l})
by *auto arith*

declare *upt_Suc*[*simp del*]

lookup1 **lemma** *nth_list_lookup1*: ⟦*braun t*; *i* < *size t*⟧ ⟹ *list t* ! *i* = *lookup1 t* (*i*+*1*)
proof(*induction t arbitrary*: *i*)
case *Leaf* **thus** *?case* **by** *simp*
next
case *Node*
thus *?case* **using** *div2_in_bounds*[*OF Node.prems*(*1*), *of i*+*1*]
by (*auto simp*: *nth_splice minus1_div2 size_list*)
qed

lemma *list_eq_map_lookup1*: *braun t* ⟹ *list t* = *map* (*lookup1 t*) [*1*..<*size t* + *1*]
by(*auto simp add*: *list_eq_iff_nth_eq size_list nth_list_lookup1*)

update1 **lemma** *size_update1*: ⟦ *braun t*; *n* ∈ {*1*.. *size t*} ⟧ ⟹ *size*(*update1 n x t*) = *size t*
proof(*induction t arbitrary*: *n*)
case *Leaf* **thus** *?case* **by** *simp*
next
case *Node* **thus** *?case* **using** *div2_in_bounds*[*OF Node.prems*] **by** *simp*
qed

lemma *braun_update1*: ⟦*braun t*; *n* ∈ {*1*.. *size t*} ⟧ ⟹ *braun*(*update1 n x t*)
proof(*induction t arbitrary*: *n*)
case *Leaf* **thus** *?case* **by** *simp*
next
case *Node* **thus** *?case*
using *div2_in_bounds*[*OF Node.prems*] **by** (*simp add*: *size_update1*)
qed

lemma *lookup1_update1*: ⟦ *braun t*; *n* ∈ {*1*.. *size t*} ⟧ ⟹
lookup1 (*update1 n x t*) *m* = (*if n*=*m then x else lookup1 t m*)
proof(*induction t arbitrary*: *m n*)
case *Leaf*
then show *?case* **by** *simp*
next
have *aux*: ⟦ *odd n*; *odd m* ⟧ ⟹ *n div 2* = (*m*::*nat*) *div 2* ⟷ *m*=*n* **for** *m n*
using *odd_two_times_div_two_succ* **by** *fastforce*
case *Node*
thus *?case* **using** *div2_in_bounds*[*OF Node.prems*] **by** (*auto simp*: *aux*)
qed

lemma *list_update1*: ⟦ *braun t*; *n* ∈ {*1*.. *size t*} ⟧ ⟹ *list*(*update1 n x t*) = (*list t*)[*n*−*1* := *x*]
by(*auto simp add*: *list_eq_map_lookup1 list_eq_iff_nth_eq lookup1_update1 size_update1 braun_update1*)

A second proof of ⟦*braun ?t*; *?n* ∈ {*1*..*size ?t*}⟧ ⟹ *list* (*update1 ?n ?x ?t*) = (*list ?t*)[*?n* − *1* := *?x*]:

lemma *diff1_eq_iff*: *n* > *0* ⟹ *n* − *Suc 0* = *m* ⟷ *n* = *m*+*1*
by *arith*

lemma *list_update_splice*:
⟦ *n* < *size xs* + *size ys*; *size ys* ≤ *size xs*; *size xs* ≤ *size ys* + *1* ⟧ ⟹
(*splice xs ys*) [*n* := *x*] =
(*if even n then splice* (*xs*[*n div 2* := *x*]) *ys else splice xs* (*ys*[*n div 2* := *x*]))
by(*induction xs ys arbitrary*: *n rule*: *splice.induct*) (*auto split*: *nat.split*)

lemma *list_update2*: ⟦ *braun t*; *n* ∈ {*1*.. *size t*} ⟧ ⟹ *list*(*update1 n x t*) = (*list t*)[*n*−*1* := *x*]
proof(*induction t arbitrary*: *n*)
case *Leaf* **thus** *?case* **by** *simp*
next
case (*Node l a r*) **thus** *?case* **using** *div2_in_bounds*[*OF Node.prems*]
by(*auto simp*: *list_update_splice diff1_eq_iff size_list split*: *nat.split*)
qed

adds **lemma** *splice_last*: **shows**
size ys ≤ *size xs* ⟹ *splice* (*xs* @ [*x*]) *ys* = *splice xs ys* @ [*x*]
and *size ys*+*1* ≥ *size xs* ⟹ *splice xs* (*ys* @ [*y*]) = *splice xs ys* @ [*y*]
by(*induction xs ys arbitrary*: *x y rule*: *splice.induct*) (*auto*)

lemma *list_add_hi*: *braun t* ⟹ *list*(*update1* (*Suc*(*size t*)) *x t*) = *list t* @ [*x*]
by(*induction t*)(*auto simp*: *splice_last size_list*)

lemma *size_add_hi*: *braun t* ⟹ *m* = *size t* ⟹ *size*(*update1* (*Suc m*) *x t*) = *size t* + *1*
by(*induction t arbitrary*: *m*)(*auto*)

lemma *braun_add_hi*: *braun t* ⟹ *braun*(*update1* (*Suc*(*size t*)) *x t*)
by(*induction t*)(*auto simp*: *size_add_hi*)

lemma *size_braun_adds*:

⟦ *braun t*; *size t* = *n* ⟧ ⟹ *size*(*adds xs n t*) = *size t* + *length xs* ∧ *braun* (*adds xs n t*)
by(*induction xs arbitrary*: *t n*)(*auto simp*: *braun_add_hi size_add_hi*)

lemma *list_adds*: ⟦ *braun t*; *size t* = *n* ⟧ ⟹ *list*(*adds xs n t*) = *list t* @ *xs*
by(*induction xs arbitrary*: *t n*)(*auto simp*: *size_braun_adds list_add_hi size_add_hi braun_add_hi*)

31.1.2 Array Implementation

interpretation *A*: *Array*
where *lookup* = λ(*t*,*l*) *n*. *lookup1 t* (*n*+*1*)
and *update* = λ*n x* (*t*,*l*). (*update1* (*n*+*1*) *x t*, *l*)
and *len* = λ(*t*,*l*). *l*
and *array* = λ*xs*. (*adds xs 0 Leaf*, *length xs*)
and *invar* = λ(*t*,*l*). *braun t* ∧ *l* = *size t*
and *list* = λ(*t*,*l*). *list t*
proof (*standard*, *goal_cases*)
case *1* **thus** *?case* **by** (*simp add*: *nth_list_lookup1 split*: *prod.splits*)
next
case *2* **thus** *?case* **by** (*simp add*: *list_update1 split*: *prod.splits*)
next
case *3* **thus** *?case* **by** (*simp add*: *size_list split*: *prod.splits*)
next
case *4* **thus** *?case* **by** (*simp add*: *list_adds*)
next
case *5* **thus** *?case* **by** (*simp add*: *braun_update1 size_update1 split*: *prod.splits*)
next
case *6* **thus** *?case* **by** (*simp add*: *size_braun_adds split*: *prod.splits*)
qed

31.2 Flexible Array

fun *add_lo* **where**
add_lo x Leaf = *Node Leaf x Leaf* |
add_lo x (*Node l a r*) = *Node* (*add_lo a r*) *x l*

fun *merge* **where**
merge Leaf r = *r* |
merge (*Node l a r*) *rr* = *Node rr a* (*merge l r*)

fun *del_lo* **where**
del_lo Leaf = *Leaf* |
del_lo (*Node l a r*) = *merge l r*

fun *del_hi* :: *nat* ⇒ *'a tree* ⇒ *'a tree* **where**
del_hi n Leaf = *Leaf* |
del_hi n (*Node l x r*) =
(*if n* = *1 then Leaf*
else if even n
then Node (*del_hi* (*n div 2*) *l*) *x r*
else Node l x (*del_hi* (*n div 2*) *r*))

31.2.1 Functional Correctness

add_lo **lemma** *list_add_lo*: *braun t* ⟹ *list* (*add_lo a t*) = *a* # *list t*
by(*induction t arbitrary*: *a*) *auto*

lemma *braun_add_lo*: *braun t* ⟹ *braun*(*add_lo x t*)
by(*induction t arbitrary*: *x*) (*auto simp add*: *list_add_lo simp flip*: *size_list*)

del_lo **lemma** *list_merge*: *braun* (*Node l x r*) ⟹ *list*(*merge l r*) = *splice* (*list l*) (*list r*)
by (*induction l r rule*: *merge.induct*) *auto*

lemma *braun_merge*: *braun* (*Node l x r*) ⟹ *braun*(*merge l r*)
by (*induction l r rule*: *merge.induct*)(*auto simp add*: *list_merge simp flip*: *size_list*)

lemma *list_del_lo*: *braun t* ⟹ *list*(*del_lo t*) = *tl* (*list t*)
by (*cases t*) (*simp_all add*: *list_merge*)

lemma *braun_del_lo*: *braun t* ⟹ *braun*(*del_lo t*)
by (*cases t*) (*simp_all add*: *braun_merge*)

del_hi **lemma** *list_Nil_iff*: *list t* = [] ⟷ *t* = *Leaf*
by(*cases t*) *simp_all*

lemma *butlast_splice*: *butlast* (*splice xs ys*) =
(*if size xs* > *size ys then splice* (*butlast xs*) *ys else splice xs* (*butlast ys*))
by(*induction xs ys rule*: *splice.induct*) (*auto*)

lemma *list_del_hi*: *braun t* ⟹ *size t* = *st* ⟹ *list*(*del_hi st t*) = *butlast*(*list t*)
apply(*induction t arbitrary*: *st*)
by(*auto simp*: *list_Nil_iff size_list butlast_splice*)

lemma *braun_del_hi*: *braun t* ⟹ *size t* = *st* ⟹ *braun*(*del_hi st t*)

apply(*induction t arbitrary*: *st*)
by(*auto simp*: *list_del_hi simp flip*: *size_list*)

31.2.2 Flexible Array Implementation

interpretation *AF*: *Array_Flex*
where *lookup* = λ(*t*,*l*) *n*. *lookup1 t* (*n*+*1*)
and *update* = λ*n x* (*t*,*l*). (*update1* (*n*+*1*) *x t*, *l*)
and *len* = λ(*t*,*l*). *l*
and *array* = λ*xs*. (*adds xs 0 Leaf*, *length xs*)
and *invar* = λ(*t*,*l*). *braun t* ∧ *l* = *size t*
and *list* = λ(*t*,*l*). *list t*
and *add_lo* = λ*x* (*t*,*l*). (*add_lo x t*, *l*+*1*)
and *del_lo* = λ(*t*,*l*). (*del_lo t*, *l*−*1*)
and *add_hi* = λ*x* (*t*,*l*). (*update1* (*Suc l*) *x t*, *l*+*1*)
and *del_hi* = λ(*t*,*l*). (*del_hi l t*, *l*−*1*)
proof (*standard*, *goal_cases*)
case *1* **thus** *?case* **by** (*simp add*: *list_add_lo split*: *prod.splits*)
next
case *2* **thus** *?case* **by** (*simp add*: *list_del_lo split*: *prod.splits*)
next
case *3* **thus** *?case* **by** (*simp add*: *list_add_hi braun_add_hi split*: *prod.splits*)
next
case *4* **thus** *?case* **by** (*simp add*: *list_del_hi split*: *prod.splits*)
next
case *5* **thus** *?case* **by** (*simp add*: *braun_add_lo list_add_lo flip*: *size_list split*: *prod.splits*)
next
case *6* **thus** *?case* **by** (*simp add*: *braun_del_lo list_del_lo flip*: *size_list split*: *prod.splits*)
next
case *7* **thus** *?case* **by** (*simp add*: *size_add_hi braun_add_hi split*: *prod.splits*)
next
case *8* **thus** *?case* **by** (*simp add*: *braun_del_hi list_del_hi flip*: *size_list split*: *prod.splits*)
qed

31.3 Faster

31.3.1 Size

fun *diff* :: *'a tree* ⇒ *nat* ⇒ *nat* **where**
diff Leaf 0 = *0* |
diff (*Node l x r*) *n* = (*if n*=*0 then 1 else if even n then diff r* (*n div 2* − *1*) *else diff l* (*n div 2*))

fun *size_fast* :: *'a tree* ⇒ *nat* **where**
size_fast Leaf = *0* |
size_fast (*Node l x r*) = (*let n* = *size_fast r in 1* + *2*∗*n* + *diff l n*)

lemma *diff*: *braun t* ⟹ *size t* : {*n*, *n* + *1*} ⟹ *diff t n* = *size t* − *n*
by(*induction t arbitrary*: *n*) *auto*

lemma *size_fast*: *braun t* ⟹ *size_fast t* = *size t*
by(*induction t*) (*auto simp add*: *Let_def diff*)

31.3.2 Initialization with 1 element

fun *braun_of_naive* :: *'a* ⇒ *nat* ⇒ *'a tree* **where**
braun_of_naive x n = (*if n=0 then Leaf*
else let m = (*n−1*) *div 2*
in if odd n then Node (*braun_of_naive x m*) *x* (*braun_of_naive x m*)
else Node (*braun_of_naive x* (*m* + *1*)) *x* (*braun_of_naive x m*))

fun *braun2_of* :: *'a* ⇒ *nat* ⇒ *'a tree* ∗ *'a tree* **where**
braun2_of x n = (*if n* = *0 then* (*Leaf*, *Node Leaf x Leaf*)
else let (*s*,*t*) = *braun2_of x* ((*n−1*) *div 2*)
in if odd n then (*Node s x s*, *Node t x s*) *else* (*Node t x s*, *Node t x t*))

definition *braun_of* :: *'a* ⇒ *nat* ⇒ *'a tree* **where**
braun_of x n = *fst* (*braun2_of x n*)

declare *braun2_of.simps* [*simp del*]

lemma *braun2_of_size_braun*: *braun2_of x n* = (*s*,*t*) ⟹ *size s* = *n* ∧ *size t* = *n+1* ∧ *braun s* ∧ *braun t*
proof(*induction x n arbitrary*: *s t rule*: *braun2_of.induct*)
case (*1 x n*)
then show *?case*
by (*auto simp*: *braun2_of.simps*[*of x n*] *split*: *prod.splits if_splits*) *presburger+*
qed

lemma *braun2_of_replicate*:
braun2_of x n = (*s*,*t*) ⟹ *list s* = *replicate n x* ∧ *list t* = *replicate* (*n+1*) *x*
proof(*induction x n arbitrary*: *s t rule*: *braun2_of.induct*)
case (*1 x n*)
have *x* # *replicate m x* = *replicate* (*m+1*) *x* **for** *m* **by** *simp*

with *1* **show** *?case*
apply (*auto simp*: *braun2_of.simps*[*of x n*] *replicate.simps*(*2*)[*of 0 x*]
simp del: *replicate.simps*(*2*) *split*: *prod.splits if_splits*)
by *presburger*+
qed

corollary *braun_braun_of*: *braun*(*braun_of x n*)
unfolding *braun_of_def* **by** (*metis eq_fst_iff braun2_of_size_braun*)

corollary *list_braun_of*: *list*(*braun_of x n*) = *replicate n x*
unfolding *braun_of_def* **by** (*metis eq_fst_iff braun2_of_replicate*)

31.3.3 Proof Infrastructure

Originally due to Thomas Sewell.

take_nths **fun** *take_nths* :: *nat* ⇒ *nat* ⇒ *'a list* ⇒ *'a list* **where**
take_nths i k [] = [] |
take_nths i k (*x* # *xs*) = (*if i* = *0 then x* # *take_nths* (*2^k* − *1*) *k xs*
else take_nths (*i* − *1*) *k xs*)

lemma *take_nths_drop*:
take_nths i k (*drop j xs*) = *take_nths* (*i* + *j*) *k xs*
by (*induct xs arbitrary*: *i j*; *simp add*: *drop_Cons split*: *nat.split*)

lemma *take_nths_00*:
take_nths 0 0 xs = *xs*
by (*induct xs*; *simp*)

lemma *splice_take_nths*:
splice (*take_nths 0* (*Suc 0*) *xs*) (*take_nths* (*Suc 0*) (*Suc 0*) *xs*) = *xs*
by (*induct xs*; *simp*)

lemma *take_nths_take_nths*:
take_nths i m (*take_nths j n xs*) = *take_nths* ((*i* ∗ *2^n*) + *j*) (*m* + *n*) *xs*
by (*induct xs arbitrary*: *i j*; *simp add*: *algebra_simps power_add*)

lemma *take_nths_empty*:
(*take_nths i k xs* = []) = (*length xs* ≤ *i*)
by (*induction xs arbitrary*: *i k*) *auto*

lemma *hd_take_nths*:
i < *length xs* ⟹ *hd*(*take_nths i k xs*) = *xs* ! *i*
by (*induction xs arbitrary*: *i k*) *auto*

lemma *take_nths_01_splice*:
⟦ *length xs = length ys ∨ length xs = length ys + 1* ⟧ ⟹
take_nths 0 (Suc 0) (splice xs ys) = xs ∧
take_nths (Suc 0) (Suc 0) (splice xs ys) = ys
by (*induct xs arbitrary*: *ys*; *case_tac ys*; *simp*)

lemma *length_take_nths_00*:
length (take_nths 0 (Suc 0) xs) = length (take_nths (Suc 0) (Suc 0) xs)
∨
length (take_nths 0 (Suc 0) xs) = length (take_nths (Suc 0) (Suc 0) xs)
+ 1
by (*induct xs*) *auto*

braun_list **fun** *braun_list* :: *'a tree ⇒ 'a list ⇒ bool* **where**
braun_list Leaf xs = (xs = []) |
braun_list (Node l x r) xs = (xs ≠ [] ∧ x = hd xs ∧
braun_list l (take_nths 1 1 xs) ∧
braun_list r (take_nths 2 1 xs))

lemma *braun_list_eq*:
braun_list t xs = (braun t ∧ xs = list t)
proof (*induct t arbitrary*: *xs*)
case *Leaf*
show *?case* **by** *simp*
next
case *Node*
show *?case*
using *length_take_nths_00*[*of xs*] *splice_take_nths*[*of xs*]
by (*auto simp*: *neq_Nil_conv Node.hyps size_list*[*symmetric*] *take_nths_01_splice*)
qed

31.3.4 Converting a list of elements into a Braun tree

fun *nodes* :: *'a tree list ⇒ 'a list ⇒ 'a tree list ⇒ 'a tree list* **where**
nodes (l#ls) (x#xs) (r#rs) = Node l x r # nodes ls xs rs |
nodes (l#ls) (x#xs) [] = Node l x Leaf # nodes ls xs [] |
nodes [] (x#xs) (r#rs) = Node Leaf x r # nodes [] xs rs |
nodes [] (x#xs) [] = Node Leaf x Leaf # nodes [] xs [] |
nodes ls [] rs = []

fun *brauns* :: *nat ⇒ 'a list ⇒ 'a tree list* **where**
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))

declare brauns.simps[simp del]

definition brauns1 :: 'a list ⇒ 'a tree where
brauns1 xs = (if xs = [] then Leaf else brauns 0 xs ! 0)

fun t_brauns :: nat ⇒ 'a list ⇒ nat where
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 * min (2^k) (length xs) + t_brauns (k+1) zs)
```

Functional correctness The proof is originally due to Thomas Sewell.

```
lemma length_nodes:
  length (nodes ls xs rs) = length xs
by (induct ls xs rs rule: nodes.induct; simp)

lemma nth_nodes:
  i < length xs ⟹ nodes ls xs rs ! i =
  Node (if i < length ls then ls ! i else Leaf) (xs ! i)
    (if i < length rs then rs ! i else Leaf)
by (induct ls xs rs arbitrary: i rule: nodes.induct;
    simp add: nth_Cons split: nat.split)

theorem length_brauns:
  length (brauns k xs) = min (length xs) (2 ^ k)
proof (induct xs arbitrary: k rule: measure_induct_rule[where f=length])
  case (less xs) thus ?case by (simp add: brauns.simps[of k xs] Let_def
length_nodes)
qed

theorem brauns_correct:
  i < min (length xs) (2 ^ k) ⟹ braun_list (brauns k xs ! i) (take_nths i k
xs)
proof (induct xs arbitrary: i k rule: measure_induct_rule[where f=length])
  case (less xs)
  have xs ≠ [] using less.prems by auto
  let ?zs = drop (2^k) xs
  let ?ts = brauns (Suc k) ?zs
```

```
  from less.hyps[of ?zs _ Suc k]
  have IH: ⟦ j = i + 2 ^ k;  i < min (length ?zs) (2 ^ (k+1)) ⟧ ⟹
    braun_list (?ts ! i) (take_nths j (Suc k) xs) for i j
    using ‹xs ≠ []› by (simp add: take_nths_drop)
  let ?xs' = take_nths i k xs
  let ?ts' = drop (2^k) ?ts
  show ?case
  proof (cases i < length ?ts ∧ ¬ i < length ?ts')
    case True
    have braun_list (brauns k xs ! i) ?xs' ⟷
          braun_list (nodes ?ts (take (2^k) xs) ?ts' ! i) ?xs'
      using ‹xs ≠ []› by (simp add: brauns.simps[of k xs] Let_def)
    also have ... ⟷ braun_list (?ts ! i) (take_nths (2^k + i) (k+1) xs)
                    ∧ braun_list Leaf (take_nths (2^(k+1) + i) (k+1) xs)
      using less.prems True
        by (clarsimp simp: nth_nodes take_nths_take_nths take_nths_empty
hd_take_nths)
   also have ... using less.prems True by (auto simp: IH take_nths_empty
length_brauns)
    finally show ?thesis .
  next
    case False
    thus ?thesis using less.prems
   by (auto simp: brauns.simps[of k xs] Let_def nth_nodes take_nths_take_nths
                  IH take_nths_empty hd_take_nths length_brauns)
  qed
qed

corollary brauns1_correct:
  braun (brauns1 xs) ∧ list (brauns1 xs) = xs
using brauns_correct[of 0 xs 0]
by (simp add: brauns1_def braun_list_eq take_nths_00)

Running Time Analysis  theorem t_brauns:
  t_brauns k xs = 4 * length xs
proof (induction xs arbitrary: k rule: measure_induct_rule[where f = length])
  case (less xs)
  show ?case
  proof cases
    assume xs = []
    thus ?thesis by(simp add: Let_def)
  next
    assume xs ≠ []
```

```
    let ?zs = drop (2^k) xs
    have t_brauns k xs = t_brauns (k+1) ?zs + 4 * min (2^k) (length xs)
     using ‹xs ≠ []› by(simp add: Let_def)
    also have ... = 4 * length ?zs + 4 * min (2^k) (length xs)
      using less[of ?zs k+1] ‹xs ≠ []›
      by (simp)
    also have ... = 4 * length xs
      by(simp)
    finally show ?case .
  qed
qed
```

31.3.5 Converting a Braun Tree into a List of Elements

The code and the proof are originally due to Thomas Sewell (except running time).

```
function list_fast_rec :: 'a tree list ⇒ 'a list where
list_fast_rec ts = (let us = filter (λt. t ≠ Leaf) ts in
  if us = [] then [] else
  map value us @ list_fast_rec (map left us @ map right us))
by (pat_completeness, auto)

lemma list_fast_rec_term1: ts ≠ [] ⟹ Leaf ∉ set ts ⟹
  sum_list (map (size o left) ts) + sum_list (map (size o right) ts) < sum_list
(map size ts)
  apply (clarsimp simp: sum_list_addf[symmetric] sum_list_map_filter')
  apply (rule sum_list_strict_mono; clarsimp?)
  apply (case_tac x; simp)
  done

lemma list_fast_rec_term: us ≠ [] ⟹ us = filter (λt. t ≠ ⟨⟩) ts ⟹
   sum_list (map (size o left) us) + sum_list (map (size o right) us) <
sum_list (map size ts)
  apply (rule order_less_le_trans, rule list_fast_rec_term1, simp_all)
  apply (rule sum_list_filter_le_nat)
  done

termination
  apply (relation measure (sum_list o map size))
   apply simp
  apply (simp add: list_fast_rec_term)
  done

declare list_fast_rec.simps[simp del]
```

definition *list_fast* :: *'a tree* ⇒ *'a list* **where**
list_fast t = *list_fast_rec* [*t*]

function *t_list_fast_rec* :: *'a tree list* ⇒ *nat* **where**
t_list_fast_rec ts = (*let us* = *filter* (λ*t*. *t* ≠ *Leaf*) *ts*
in length ts + (*if us* = [] *then 0 else*
5 ∗ *length us* + *t_list_fast_rec* (*map left us* @ *map right us*)))
by (*pat_completeness*, *auto*)

termination
apply (*relation measure* (*sum_list o map size*))
apply *simp*
apply (*simp add*: *list_fast_rec_term*)
done

declare *t_list_fast_rec.simps*[*simp del*]

Functional Correctness **lemma** *list_fast_rec_all_Leaf*:
∀ *t* ∈ *set ts*. *t* = *Leaf* ⟹ *list_fast_rec ts* = []
by (*simp add*: *filter_empty_conv list_fast_rec.simps*)

lemma *take_nths_eq_single*:
length xs − *i* < *2*^*n* ⟹ *take_nths i n xs* = *take 1* (*drop i xs*)
by (*induction xs arbitrary*: *i n*; *simp add*: *drop_Cons'*)

lemma *braun_list_Nil*:
braun_list t [] = (*t* = *Leaf*)
by (*cases t*; *simp*)

lemma *braun_list_not_Nil*: *xs* ≠ [] ⟹
braun_list t xs =
(∃ *l x r*. *t* = *Node l x r* ∧ *x* = *hd xs* ∧
braun_list l (*take_nths 1 1 xs*) ∧
braun_list r (*take_nths 2 1 xs*))
by(*cases t*; *simp*)

theorem *list_fast_rec_correct*:
⟦ *length ts* = *2* ^ *k*; ∀ *i* < *2* ^ *k*. *braun_list* (*ts* ! *i*) (*take_nths i k xs*) ⟧
⟹ *list_fast_rec ts* = *xs*
proof (*induct xs arbitrary*: *k ts rule*: *measure_induct_rule*[**where** *f*=*length*])
case (*less xs*)
show *?case*

proof (*cases length xs < 2 ^ k*)
case *True*
from *less.prems True* **have** *filter*:
$\exists n.\ ts = map\ (\lambda x.\ Node\ Leaf\ x\ Leaf)\ xs\ @\ replicate\ n\ Leaf$
apply (*rule_tac x=length ts − length xs* **in** *exI*)
apply (*clarsimp simp*: *list_eq_iff_nth_eq*)
apply(*auto simp*: *nth_append braun_list_not_Nil take_nths_eq_single braun_list_Nil hd_drop_conv_nth*)
done
thus *?thesis*
by (*clarsimp simp*: *list_fast_rec.simps*[*of ts*] *o_def list_fast_rec_all_Leaf Let_def*)
next
case *False*
with *less.prems*(*2*) **have** ∗:
$\forall i < 2\ \hat{}\ k.\ ts\ !\ i \neq Leaf$
$\wedge\ value\ (ts\ !\ i) = xs\ !\ i$
$\wedge\ braun\_list\ (left\ (ts\ !\ i))\ (take\_nths\ (i + 2\ \hat{}\ k)\ (Suc\ k)\ xs)$
$\wedge\ (\forall ys.\ ys = take\_nths\ (i + 2 * 2\ \hat{}\ k)\ (Suc\ k)\ xs$
$\longrightarrow braun\_list\ (right\ (ts\ !\ i))\ ys)$
by (*auto simp*: *take_nths_empty hd_take_nths braun_list_not_Nil take_nths_take_nths algebra_simps*)
have *1*: *map value ts = take* (*2 ^ k*) *xs*
using *less.prems*(*1*) *False* **by** (*simp add*: *list_eq_iff_nth_eq* ∗)
have *2*: *list_fast_rec* (*map left ts @ map right ts*) = *drop* (*2 ^ k*) *xs*
using *less.prems*(*1*) *False*
by (*auto intro*!: *Nat.diff_less less.hyps*[**where** *k*= *Suc k*]
simp: *nth_append* ∗ *take_nths_drop algebra_simps*)
from *less.prems*(*1*) *False* **show** *?thesis*
by (*auto simp*: *list_fast_rec.simps*[*of ts*] *1 2 Let_def* ∗ *all_set_conv_all_nth*)
qed
qed

corollary *list_fast_correct*:
braun t ⟹ *list_fast t = list t*
by (*simp add*: *list_fast_def take_nths_00 braun_list_eq list_fast_rec_correct*[**where** *k=0*])

Running Time Analysis **lemma** *sum_tree_list_children*: ∀ *t* ∈ *set ts. t* ≠ *Leaf* ⟹
$(\sum t \leftarrow ts.\ k * size\ t) = (\sum t \leftarrow map\ left\ ts\ @\ map\ right\ ts.\ k * size\ t) + k * length\ ts$
by(*induction ts*)(*auto simp add*: *neq_Leaf_iff algebra_simps*)

theorem *t_list_fast_rec_ub*:
t_list_fast_rec ts ≤ *sum_list* (*map* (λ*t*. 7∗*size t* + 1) *ts*)
proof (*induction ts rule*: *measure_induct_rule*[**where** *f*=*sum_list o map size*])
case (*less ts*)
let *?us* = *filter* (λ*t*. *t* ≠ *Leaf*) *ts*
show *?case*
proof *cases*
assume *?us* = []
thus *?thesis* **using** *t_list_fast_rec.simps*[*of ts*]
by(*simp add*: *Let_def sum_list_Suc*)
next
assume *?us* ≠ []
let *?children* = *map left ?us* @ *map right ?us*
have *t_list_fast_rec ts* = *t_list_fast_rec ?children* + 5 ∗ *length ?us* + *length ts*
using ‹*?us* ≠ []› *t_list_fast_rec.simps*[*of ts*] **by**(*simp add*: *Let_def*)
also have ... ≤ (∑ *t*←*?children*. 7 ∗ *size t* + 1) + 5 ∗ *length ?us* + *length ts*
using *less*[*of ?children*] *list_fast_rec_term*[*of ?us*] ‹*?us* ≠ []›
by (*simp*)
also have ... = (∑ *t*←*?children*. 7∗*size t*) + 7 ∗ *length ?us* + *length ts*
by(*simp add*: *sum_list_Suc o_def*)
also have ... = (∑ *t*←*?us*. 7∗*size t*) + *length ts*
by(*simp add*: *sum_tree_list_children*)
also have ... ≤ (∑ *t*←*ts*. 7∗*size t*) + *length ts*
by(*simp add*: *sum_list_filter_le_nat*)
also have ... = (∑ *t*←*ts*. 7 ∗ *size t* + 1)
by(*simp add*: *sum_list_Suc*)
finally show *?case* **.**
qed
qed

end

32 Tries via Functions

theory *Trie_Fun*
imports
Set_Specs
begin

A trie where each node maps a key to sub-tries via a function. Nice abstract model. Not efficient because of the function space.

datatype *'a trie = Nd bool 'a ⇒ 'a trie option*

fun *isin* :: *'a trie ⇒ 'a list ⇒ bool* **where**
isin (Nd b m) [] = b |
isin (Nd b m) (k # xs) = (case m k of None ⇒ False | Some t ⇒ isin t xs)

fun *insert* :: *('a::linorder) list ⇒ 'a trie ⇒ 'a trie* **where**
insert [] (Nd b m) = Nd True m |
insert (x#xs) (Nd b m) =
Nd b (m(x := Some(insert xs (case m x of None ⇒ Nd False (λ_. None) | Some t ⇒ t))))

fun *delete* :: *('a::linorder) list ⇒ 'a trie ⇒ 'a trie* **where**
delete [] (Nd b m) = Nd False m |
delete (x#xs) (Nd b m) = Nd b
(case m x of
None ⇒ m |
Some t ⇒ m(x := Some(delete xs t)))

The actual definition of *set* is a bit cryptic but canonical, to enable primrec to prove termination:

primrec *set* :: *'a trie ⇒ 'a list set* **where**
set (Nd b m) = (if b then {[]} else {}) ∪
(⋃a. case (map_option set o m) a of None ⇒ {} | Some t ⇒ (#) a ‘ t)

This is the more human-readable version:

lemma *set_Nd*:
set (Nd b m) =
(if b then {[]} else {}) ∪
(⋃a. case m a of None ⇒ {} | Some t ⇒ (#) a ‘ set t)
by (*auto simp*: *split*: *option.splits*)

lemma *isin_set*: *isin t xs = (xs ∈ set t)*
apply(*induction t xs rule*: *isin.induct*)
apply (*auto split*: *option.split*)
done

lemma *set_insert*: *set (insert xs t) = set t ∪ {xs}*
proof(*induction xs t rule*: *insert.induct*)
case *1* **thus** *?case* **by** *simp*
next

```
  case 2
  thus ?case
    apply(simp)
    apply(subst set_eq_iff)
    apply(auto split!: if_splits option.splits)
     apply fastforce
    by (metis imageI option.sel)
qed

lemma set_delete: set (delete xs t) = set t − {xs}
proof(induction xs t rule: delete.induct)
  case 1 thus ?case by (force split: option.splits)
next
  case 2
  thus ?case
    apply (auto simp add: image_iff split!: if_splits option.splits)
       apply blast
      apply (metis insertE insertI2 insert_Diff_single option.inject)
     apply blast
    by (metis insertE insertI2 insert_Diff_single option.inject)
qed

interpretation S: Set
where empty = Nd False (λ_. None) and isin = isin and insert = insert
and delete = delete
and set = set and invar = λ_. True
proof (standard, goal_cases)
  case 1 show ?case by (simp)
next
  case 2 thus ?case by(simp add: isin_set)
next
  case 3 thus ?case by(simp add: set_insert)
next
  case 4 thus ?case by(simp add: set_delete)
qed (rule TrueI)+

end
```

33 Tries via Search Trees

```
theory Trie_Map
imports
  RBT_Map
```

Trie_Fun
begin

An implementation of tries based on maps implemented by red-black trees. Works for any kind of search tree.

Implementation of map:

type_synonym *'a mapi = 'a rbt*

datatype *'a trie_map = Nd bool ('a * 'a trie_map) mapi*

In principle one should be able to given an implementation of tries once and for all for any map implementation and not just for a specific one (RBT) as done here. But because the map (*'a rbt*) is used in a datatype, the HOL type system does not support this.

However, the development below works verbatim for any map implementation, eg *Tree_Map*, and not just *RBT_Map*, except for the termination lemma *lookup_size*.

lemma *lookup_size*[*termination_simp*]:
fixes *t* :: (*'a*::*linorder* * *'a trie_map*) *rbt*
shows *lookup t a = Some b* ⟹ *size b* < *Suc* (*size_tree* (λ*ab*. *Suc* (*size* (*snd ab*))) (λ*x*. *0*) *t*)
apply(*induction t a rule*: *lookup.induct*)
apply(*auto split*: *if_splits*)
done

fun *isin* :: (*'a*::*linorder*) *trie_map* ⇒ *'a list* ⇒ *bool* **where**
isin (*Nd b m*) [] = *b* |
isin (*Nd b m*) (*x* # *xs*) = (*case lookup m x of None* ⇒ *False* | *Some t* ⇒ *isin t xs*)

fun *insert* :: (*'a*::*linorder*) *list* ⇒ *'a trie_map* ⇒ *'a trie_map* **where**
insert [] (*Nd b m*) = *Nd True m* |
insert (*x*#*xs*) (*Nd b m*) =
Nd b (*update x* (*insert xs* (*case lookup m x of None* ⇒ *Nd False Leaf* | *Some t* ⇒ *t*)) *m*)

fun *delete* :: (*'a*::*linorder*) *list* ⇒ *'a trie_map* ⇒ *'a trie_map* **where**
delete [] (*Nd b m*) = *Nd False m* |
delete (*x*#*xs*) (*Nd b m*) = *Nd b*
(*case lookup m x of*
None ⇒ *m* |
Some t ⇒ *update x* (*delete xs t*) *m*)

33.1 Correctness

Proof by stepwise refinement. First abstract to type *'a trie*.

fun *abs* :: *'a*::*linorder trie_map* ⇒ *'a trie* **where**
abs (*Nd b t*) = *Trie_Fun.Nd b* (λ*a*. *map_option abs* (*lookup t a*))

fun *invar* :: (*'a*::*linorder*)*trie_map* ⇒ *bool* **where**
invar (*Nd b m*) = (*M.invar m* ∧ (∀ *a t*. *lookup m a* = *Some t* ⟶ *invar t*))

lemma *isin_abs*: *isin t xs* = *Trie_Fun.isin* (*abs t*) *xs*
apply(*induction t xs rule*: *isin.induct*)
apply(*auto split*: *option.split*)
done

lemma *abs_insert*: *invar t* ⟹ *abs*(*insert xs t*) = *Trie_Fun.insert xs* (*abs t*)
apply(*induction xs t rule*: *insert.induct*)
apply(*auto simp*: *M.map_specs RBT_Set.empty_def*[*symmetric*] *split*: *option.split*)
done

lemma *abs_delete*: *invar t* ⟹ *abs*(*delete xs t*) = *Trie_Fun.delete xs* (*abs t*)
apply(*induction xs t rule*: *delete.induct*)
apply(*auto simp*: *M.map_specs split*: *option.split*)
done

lemma *invar_insert*: *invar t* ⟹ *invar* (*insert xs t*)
apply(*induction xs t rule*: *insert.induct*)
apply(*auto simp*: *M.map_specs RBT_Set.empty_def*[*symmetric*] *split*: *option.split*)
done

lemma *invar_delete*: *invar t* ⟹ *invar* (*delete xs t*)
apply(*induction xs t rule*: *delete.induct*)
apply(*auto simp*: *M.map_specs split*: *option.split*)
done

Overall correctness w.r.t. the *Set* ADT:

interpretation *S2*: *Set*
where *empty* = *Nd False Leaf* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*
and *set* = *set o abs* **and** *invar* = *invar*

proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** (*simp*)
next
case *2* **thus** *?case* **by** (*simp add*: *isin_set isin_abs*)
next
case *3* **thus** *?case* **by** (*simp add*: *set_insert abs_insert*)
next
case *4* **thus** *?case* **by** (*simp add*: *set_delete abs_delete*)
next
case *5* **thus** *?case* **by** (*simp add*: *M.map_specs RBT_Set.empty_def*[*symmetric*])
next
case *6* **thus** *?case* **by** (*simp add*: *invar_insert*)
next
case *7* **thus** *?case* **by** (*simp add*: *invar_delete*)
qed

end

34 Binary Tries and Patricia Tries

theory *Tries_Binary*
imports *Set_Specs*
begin

hide_const (**open**) *insert*

declare *Let_def*[*simp*]

fun *sel2* :: *bool* ⇒ *'a* ∗ *'a* ⇒ *'a* **where**
sel2 b (*a1*,*a2*) = (*if b then a2 else a1*)

fun *mod2* :: (*'a* ⇒ *'a*) ⇒ *bool* ⇒ *'a* ∗ *'a* ⇒ *'a* ∗ *'a* **where**
mod2 f b (*a1*,*a2*) = (*if b then* (*a1*,*f a2*) *else* (*f a1*,*a2*))

34.1 Trie

datatype *trie* = *Lf* | *Nd bool trie* ∗ *trie*

fun *isin* :: *trie* ⇒ *bool list* ⇒ *bool* **where**
isin Lf ks = *False* |
isin (*Nd b lr*) *ks* =
(*case ks of*
[] ⇒ *b* |

```
  k#ks ⇒ isin (sel2 k lr) ks)

fun insert :: bool list ⇒ trie ⇒ trie where
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)

lemma isin_insert: isin (insert as t) bs = (as = bs ∨ isin t bs)
apply(induction as t arbitrary: bs rule: insert.induct)
apply (auto split: list.splits if_splits)
done
```

A simple implementation of delete; does not shrink the trie!

```
fun delete0 :: bool list ⇒ trie ⇒ trie where
delete0 ks Lf = Lf |
delete0 ks (Nd b lr) =
  (case ks of
    [] ⇒ Nd False lr |
    k#ks' ⇒ Nd b (mod2 (delete0 ks') k lr))

lemma isin_delete0: isin (delete0 as t) bs = (as ≠ bs ∧ isin t bs)
apply(induction as t arbitrary: bs rule: delete0.induct)
apply (auto split: list.splits if_splits)
done
```

Now deletion with shrinking:

```
fun node :: bool ⇒ trie * trie ⇒ trie where
node b lr = (if ¬ b ∧ lr = (Lf,Lf) then Lf else Nd b lr)

fun delete :: bool list ⇒ trie ⇒ trie where
delete ks Lf = Lf |
delete ks (Nd b lr) =
  (case ks of
    [] ⇒ node False lr |
    k#ks' ⇒ node b (mod2 (delete ks') k lr))

lemma isin_delete: isin (delete as t) bs = (as ≠ bs ∧ isin t bs)
apply(induction as t arbitrary: bs rule: delete.induct)
 apply simp
apply (auto split: list.splits if_splits)
  apply (metis isin.simps(1))
 apply (metis isin.simps(1))
  done
```

definition *set_trie* :: *trie* ⇒ *bool list set* **where**
set_trie t = {*xs*. *isin t xs*}

lemma *set_trie_insert*: *set_trie*(*insert xs t*) = *set_trie t* ∪ {*xs*}
by(*auto simp add*: *isin_insert set_trie_def*)

interpretation *S*: *Set*
where *empty* = *Lf* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*
and *set* = *set_trie* **and** *invar* = λ*t*. *True*
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** (*simp add*: *set_trie_def*)
next
  case *2* **thus** *?case* **by**(*simp add*: *set_trie_def*)
next
  case *3* **thus** *?case* **by**(*auto simp*: *set_trie_insert*)
next
  case *4* **thus** *?case* **by**(*auto simp*: *isin_delete set_trie_def*)
qed (*rule TrueI*)+

34.2 Patricia Trie

datatype *ptrie* = *LfP* | *NdP bool list bool ptrie* ∗ *ptrie*

fun *isinP* :: *ptrie* ⇒ *bool list* ⇒ *bool* **where**
isinP LfP ks = *False* |
isinP (*NdP ps b lr*) *ks* =
  (*let n* = *length ps in*
   if ps = *take n ks*
   then case drop n ks of [] ⇒ *b* | *k*#*ks*′ ⇒ *isinP* (*sel2 k lr*) *ks*′
   else False)

fun *split* **where**
split [] *ys* = ([],[],*ys*) |
split xs [] = ([],*xs*,[]) |
split (*x*#*xs*) (*y*#*ys*) =
  (*if x*≠*y then* ([],*x*#*xs*,*y*#*ys*)
   else let (*ps*,*xs*′,*ys*′) = *split xs ys in* (*x*#*ps*,*xs*′,*ys*′))

lemma *mod2_cong*[*fundef_cong*]:
  ⟦ *lr* = *lr*′; *k* = *k*′; ⋀*a b*. *lr*′=(*a*,*b*) ⟹ *f* (*a*) = *f*′ (*a*) ; ⋀*a b*. *lr*′=(*a*,*b*)
⟹ *f* (*b*) = *f*′ (*b*) ⟧

$\Longrightarrow$ *mod2 f k lr*= *mod2 f' k' lr'*
by(*cases lr*, *cases lr'*, *auto*)

fun *insertP* :: *bool list* ⇒ *ptrie* ⇒ *ptrie* **where**
insertP ks LfP = *NdP ks True* (*LfP*,*LfP*) |
insertP ks (*NdP ps b lr*) =
(*case split ks ps of*
(*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*)) |
(*qs*,*k*#*ks'*,[]) ⇒
NdP ps b (*mod2* (*insertP ks'*) *k lr*) |
(*qs*,[],*p*#*ps'*) ⇒
let t = *NdP ps' b lr in*
NdP qs True (*if p then* (*LfP*,*t*) *else* (*t*,*LfP*)) |
(*qs*,[],[]) ⇒ *NdP ps True lr*)

fun *nodeP* :: *bool list* ⇒ *bool* ⇒ *ptrie* ∗ *ptrie* ⇒ *ptrie* **where**
nodeP ps b lr = (*if* ¬ *b* ∧ *lr* = (*LfP*,*LfP*) *then LfP else NdP ps b lr*)

fun *deleteP* :: *bool list* ⇒ *ptrie* ⇒ *ptrie* **where**
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*)

34.2.1 Functional Correctness

First step: *ptrie* implements *trie* via the abstraction function *abs_ptrie*:

fun *prefix_trie* :: *bool list* ⇒ *trie* ⇒ *trie* **where**
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*)))

fun *abs_ptrie* :: *ptrie* ⇒ *trie* **where**
abs_ptrie LfP = *Lf* |
abs_ptrie (*NdP ps b* (*l*,*r*)) = *prefix_trie ps* (*Nd b* (*abs_ptrie l*, *abs_ptrie r*))

Correctness of *isinP*:

lemma *isin_prefix_trie*:
isin (*prefix_trie ps t*) *ks*

$= (ps = take\ (length\ ps)\ ks \wedge isin\ t\ (drop\ (length\ ps)\ ks))$
apply(*induction ps arbitrary*: *ks*)
apply(*auto split*: *list.split*)
done

lemma *isinP*:
$isinP\ t\ ks = isin\ (abs\_ptrie\ t)\ ks$
apply(*induction t arbitrary*: *ks rule*: *abs_ptrie.induct*)
apply(*auto simp*: *isin_prefix_trie split*: *list.split*)
done

Correctness of *insertP*:

lemma *prefix_trie_Lfs*: *prefix_trie ks* (*Nd True* (*Lf*,*Lf*)) = *insert ks Lf*
apply(*induction ks*)
apply *auto*
done

lemma *insert_prefix_trie_same*:
$insert\ ps\ (prefix\_trie\ ps\ (Nd\ b\ lr)) = prefix\_trie\ ps\ (Nd\ True\ lr)$
apply(*induction ps*)
apply *auto*
done

lemma *insert_append*: *insert* (*ks* @ *ks*′) (*prefix_trie ks t*) = *prefix_trie ks* (*insert ks*′ *t*)
apply(*induction ks*)
apply *auto*
done

lemma *prefix_trie_append*: *prefix_trie* (*ps* @ *qs*) *t* = *prefix_trie ps* (*prefix_trie qs t*)
apply(*induction ps*)
apply *auto*
done

lemma *split_if*: *split ks ps* = (*qs*, *ks*′, *ps*′) ⟹
$ks = qs\ @\ ks' \wedge ps = qs\ @\ ps' \wedge (ks' \neq [] \wedge ps' \neq [] \longrightarrow hd\ ks' \neq hd\ ps')$
apply(*induction ks ps arbitrary*: *qs ks*′ *ps*′ *rule*: *split.induct*)
apply(*auto split*: *prod.splits if_splits*)
done

lemma *abs_ptrie_insertP*:
$abs\_ptrie\ (insertP\ ks\ t) = insert\ ks\ (abs\_ptrie\ t)$
apply(*induction t arbitrary*: *ks*)

apply(*auto simp*: *prefix_trie_Lfs insert_prefix_trie_same insert_append prefix_trie_append*
dest!: *split_if split*: *list.split prod.split if_splits*)
done

Correctness of *deleteP*:

lemma *prefix_trie_Lf*: *prefix_trie xs t* = *Lf* ⟷ *xs* = [] ∧ *t* = *Lf*
by(*cases xs*)(*auto*)

lemma *abs_ptrie_Lf*: *abs_ptrie t* = *Lf* ⟷ *t* = *LfP*
by(*cases t*) (*auto simp*: *prefix_trie_Lf*)

lemma *delete_prefix_trie*:
delete xs (*prefix_trie xs* (*Nd b* (*l*,*r*)))
= (*if* (*l*,*r*) = (*Lf*,*Lf*) *then Lf else prefix_trie xs* (*Nd False* (*l*,*r*)))
by(*induction xs*)(*auto simp*: *prefix_trie_Lf*)

lemma *delete_append_prefix_trie*:
delete (*xs* @ *ys*) (*prefix_trie xs t*)
= (*if delete ys t* = *Lf then Lf else prefix_trie xs* (*delete ys t*))
by(*induction xs*)(*auto simp*: *prefix_trie_Lf*)

lemma *delete_abs_ptrie*:
delete ks (*abs_ptrie t*) = *abs_ptrie* (*deleteP ks t*)
apply(*induction t arbitrary*: *ks*)
apply(*auto simp*: *delete_prefix_trie delete_append_prefix_trie*
prefix_trie_append prefix_trie_Lf abs_ptrie_Lf
dest!: *split_if split*: *if_splits list.split prod.split*)
done

The overall correctness proof. Simply composes correctness lemmas.

definition *set_ptrie* :: *ptrie* ⇒ *bool list set* **where**
set_ptrie = *set_trie o abs_ptrie*

lemma *set_ptrie_insertP*: *set_ptrie* (*insertP xs t*) = *set_ptrie t* ∪ {*xs*}
by(*simp add*: *abs_ptrie_insertP set_trie_insert set_ptrie_def*)

interpretation *SP*: *Set*
where *empty* = *LfP* **and** *isin* = *isinP* **and** *insert* = *insertP* **and** *delete* = *deleteP*
and *set* = *set_ptrie* **and** *invar* = λ*t*. *True*
proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** (*simp add*: *set_ptrie_def set_trie_def*)
next

case *2* **thus** *?case* **by**(*simp add*: *isinP set_ptrie_def set_trie_def*)
next
case *3* **thus** *?case* **by** (*auto simp*: *set_ptrie_insertP*)
next
case *4* **thus** *?case*
by(*auto simp*: *isin_delete set_ptrie_def set_trie_def simp flip*: *delete_abs_ptrie*)
qed (*rule TrueI*)+

end

35 Common Basis Theory

theory *Base_FDS*
imports *HOL−Library.Pattern_Aliases*
begin

declare *Let_def* [*simp*]

Lemma *size_prod_measure*, when declared with the *measure_function* attribute, enables *fun* to prove termination of a larger class of functions automatically. By default, *fun* only tries lexicographic combinations of the sizes of the parameters. With *size_prod_measure* enabled it also tries measures based on the sum of the sizes of different parameters.

To alert the reader whenever such a more subtle termination proof is taking place the lemma is not enabled all the time but only when it is needed.

lemma *size_prod_measure*:
is_measure f ⟹ *is_measure g* ⟹ *is_measure* (*size_prod f g*)
by (*rule is_measure_trivial*)

end

36 Priority Queue Specifications

theory *Priority_Queue_Specs*
imports *HOL−Library.Multiset*
begin

Priority queue interface + specification:

locale *Priority_Queue* =
fixes *empty* :: *'q*
and *is_empty* :: *'q* ⇒ *bool*
and *insert* :: *'a*::*linorder* ⇒ *'q* ⇒ *'q*

and *get_min* :: *'q* ⇒ *'a*
and *del_min* :: *'q* ⇒ *'q*
and *invar* :: *'q* ⇒ *bool*
and *mset* :: *'q* ⇒ *'a multiset*
assumes *mset_empty*: *mset empty* = {#}
and *is_empty*: *invar q* ⟹ *is_empty q* = (*mset q* = {#})
and *mset_insert*: *invar q* ⟹ *mset* (*insert x q*) = *mset q* + {#*x*#}
and *mset_del_min*: *invar q* ⟹ *mset q* ≠ {#} ⟹
mset (*del_min q*) = *mset q* − {# *get_min q* #}
and *mset_get_min*: *invar q* ⟹ *mset q* ≠ {#} ⟹ *get_min q* = *Min_mset* (*mset q*)
and *invar_empty*: *invar empty*
and *invar_insert*: *invar q* ⟹ *invar* (*insert x q*)
and *invar_del_min*: *invar q* ⟹ *mset q* ≠ {#} ⟹ *invar* (*del_min q*)

Extend locale with *merge*. Need to enforce that *'q* is the same in both locales.

locale *Priority_Queue_Merge* = *Priority_Queue* **where** *empty* = *empty* **for** *empty* :: *'q* +
fixes *merge* :: *'q* ⇒ *'q* ⇒ *'q*
assumes *mset_merge*: ⟦ *invar q1*; *invar q2* ⟧ ⟹ *mset* (*merge q1 q2*) = *mset q1* + *mset q2*
and *invar_merge*: ⟦ *invar q1*; *invar q2* ⟧ ⟹ *invar* (*merge q1 q2*)

end

37 Leftist Heap

theory *Leftist_Heap*
imports
Base_FDS
Tree2
Priority_Queue_Specs
Complex_Main
begin

fun *mset_tree* :: (*'a*,*'b*) *tree* ⇒ *'a multiset* **where**
mset_tree Leaf = {#} |
mset_tree (*Node l a _ r*) = {#*a*#} + *mset_tree l* + *mset_tree r*

type_synonym *'a lheap* = (*'a*,*nat*)*tree*

fun *rank* :: *'a lheap* ⇒ *nat* **where**
rank Leaf = *0* |

```
rank (Node _ _ _ r) = rank r + 1

fun rk :: 'a lheap ⇒ nat where
rk Leaf = 0 |
rk (Node _ _ n _) = n
```

The invariants:

```
fun (in linorder) heap :: ('a,'b) tree ⇒ bool where
heap Leaf = True |
heap (Node l m _ r) =
  (heap l ∧ heap r ∧ (∀ x ∈ set_mset(mset_tree l + mset_tree r). m ≤ x))

fun ltree :: 'a lheap ⇒ bool where
ltree Leaf = True |
ltree (Node l a n r) =
 (n = rank r + 1 ∧ rank l ≥ rank r ∧ ltree l & ltree r)

definition node :: 'a lheap ⇒ 'a ⇒ 'a lheap ⇒ 'a lheap where
node l a r =
 (let rl = rk l; rr = rk r
  in if rl ≥ rr then Node l a (rr+1) r else Node r a (rl+1) l)

fun get_min :: 'a lheap ⇒ 'a where
get_min(Node l a n r) = a
```

For function *merge*:

```
unbundle pattern_aliases
declare size_prod_measure[measure_function]

fun merge :: 'a::ord lheap ⇒ 'a lheap ⇒ 'a lheap where
merge Leaf t2 = t2 |
merge t1 Leaf = t1 |
merge (Node l1 a1 n1 r1 =: t1) (Node l2 a2 n2 r2 =: t2) =
   (if a1 ≤ a2 then node l1 a1 (merge r1 t2)
    else node l2 a2 (merge t1 r2))

lemma merge_code: merge t1 t2 = (case (t1,t2) of
  (Leaf, _) ⇒ t2 |
  (_, Leaf) ⇒ t1 |
  (Node l1 a1 n1 r1, Node l2 a2 n2 r2) ⇒
    if a1 ≤ a2 then node l1 a1 (merge r1 t2) else node l2 a2 (merge t1 r2))
by(induction t1 t2 rule: merge.induct) (simp_all split: tree.split)

hide_const (open) insert
```

definition *insert* :: *'a::ord ⇒ 'a lheap ⇒ 'a lheap* **where**
insert x t = merge (Node Leaf x 1 Leaf) t

fun *del_min* :: *'a::ord lheap ⇒ 'a lheap* **where**
del_min Leaf = Leaf |
del_min (Node l x n r) = merge l r

37.1 Lemmas

lemma *mset_tree_empty*: *mset_tree t = {#} ⟷ t = Leaf*
by(*cases t*) *auto*

lemma *rk_eq_rank*[*simp*]: *ltree t ⟹ rk t = rank t*
by(*cases t*) *auto*

lemma *ltree_node*: *ltree (node l a r) ⟷ ltree l ∧ ltree r*
by(*auto simp add*: *node_def*)

lemma *heap_node*: *heap (node l a r) ⟷*
heap l ∧ heap r ∧ (∀ x ∈ set_mset(*mset_tree l + mset_tree r*). *a ≤ x*)
by(*auto simp add*: *node_def*)

37.2 Functional Correctness

lemma *mset_merge*: *mset_tree (merge h1 h2) = mset_tree h1 + mset_tree h2*
by (*induction h1 h2 rule*: *merge.induct*) (*auto simp add*: *node_def ac_simps*)

lemma *mset_insert*: *mset_tree (insert x t) = mset_tree t + {#x#}*
by (*auto simp add*: *insert_def mset_merge*)

lemma *get_min*: *⟦ heap h; h ≠ Leaf ⟧ ⟹ get_min h = Min_mset (mset_tree h)*
by (*induction h*) (*auto simp add*: *eq_Min_iff*)

lemma *mset_del_min*: *mset_tree (del_min h) = mset_tree h − {# get_min h #}*
by (*cases h*) (*auto simp*: *mset_merge*)

lemma *ltree_merge*: *⟦ ltree l; ltree r ⟧ ⟹ ltree (merge l r)*
proof(*induction l r rule*: *merge.induct*)
case (*3 l1 a1 n1 r1 l2 a2 n2 r2*)
show *?case* (**is** *ltree*(*merge ?t1 ?t2*))

```
  proof cases
    assume a1 ≤ a2
   hence ltree (merge ?t1 ?t2) = ltree (node l1 a1 (merge r1 ?t2)) by simp
    also have ... = (ltree l1 ∧ ltree(merge r1 ?t2))
      by(simp add: ltree_node)
    also have ... using 3.prems 3.IH(1)[OF ‹a1 ≤ a2›] by (simp)
    finally show ?thesis .
  next
    assume ¬ a1 ≤ a2
    thus ?thesis using 3 by(simp)(auto simp: ltree_node)
  qed
qed simp_all

lemma heap_merge: ⟦ heap l; heap r ⟧ ⟹ heap (merge l r)
proof(induction l r rule: merge.induct)
  case 3 thus ?case by(auto simp: heap_node mset_merge ball_Un)
qed simp_all

lemma ltree_insert: ltree t ⟹ ltree(insert x t)
by(simp add: insert_def ltree_merge del: merge.simps split: tree.split)

lemma heap_insert: heap t ⟹ heap(insert x t)
by(simp add: insert_def heap_merge del: merge.simps split: tree.split)

lemma ltree_del_min: ltree t ⟹ ltree(del_min t)
by(cases t)(auto simp add: ltree_merge simp del: merge.simps)

lemma heap_del_min: heap t ⟹ heap(del_min t)
by(cases t)(auto simp add: heap_merge simp del: merge.simps)
```

Last step of functional correctness proof: combine all the above lemmas to show that leftist heaps satisfy the specification of priority queues with merge.

```
interpretation lheap: Priority_Queue_Merge
where empty = Leaf and is_empty = λh. h = Leaf
and insert = insert and del_min = del_min
and get_min = get_min and merge = merge
and invar = λh. heap h ∧ ltree h and mset = mset_tree
proof(standard, goal_cases)
  case 1 show ?case by simp
next
  case (2 q) show ?case by (cases q) auto
next
  case 3 show ?case by(rule mset_insert)
```

next
case *4* **show** *?case* **by**(*rule mset_del_min*)
next
case *5* **thus** *?case* **by**(*simp add*: *get_min mset_tree_empty*)
next
case *6* **thus** *?case* **by**(*simp*)
next
case *7* **thus** *?case* **by**(*simp add*: *heap_insert ltree_insert*)
next
case *8* **thus** *?case* **by**(*simp add*: *heap_del_min ltree_del_min*)
next
case *9* **thus** *?case* **by** (*simp add*: *mset_merge*)
next
case *10* **thus** *?case* **by** (*simp add*: *heap_merge ltree_merge*)
qed

37.3 Complexity

lemma *pow2_rank_size1*: *ltree t* $\Longrightarrow$ *2* ^ *rank t* $\leq$ *size1 t*
proof(*induction t*)
case *Leaf* **show** *?case* **by** *simp*
next
case (*Node l a n r*)
hence *rank r* $\leq$ *rank l* **by** *simp*
hence $*$: (*2*::*nat*) ^ *rank r* $\leq$ *2* ^ *rank l* **by** *simp*
have (*2*::*nat*) ^ *rank* $\langle l, a, n, r\rangle$ = *2* ^ *rank r* + *2* ^ *rank r*
by(*simp add*: *mult_2*)
also have ... $\leq$ *size1 l* + *size1 r*
using *Node* $*$ **by** (*simp del*: *power_increasing_iff*)
also have ... = *size1* $\langle l, a, n, r\rangle$ **by** *simp*
finally show *?case* **.**
qed

Explicit termination argument: sum of sizes

fun *t_merge* :: *'a*::*ord lheap* $\Rightarrow$ *'a lheap* $\Rightarrow$ *nat* **where**
t_merge Leaf t2 = *1* |
t_merge t2 Leaf = *1* |
t_merge (*Node l1 a1 n1 r1* =: *t1*) (*Node l2 a2 n2 r2* =: *t2*) =
(*if a1* $\leq$ *a2 then 1* + *t_merge r1 t2*
else 1 + *t_merge t1 r2*)

definition *t_insert* :: *'a*::*ord* $\Rightarrow$ *'a lheap* $\Rightarrow$ *nat* **where**
t_insert x t = *t_merge* (*Node Leaf x 1 Leaf*) *t*

```
fun t_del_min :: 'a::ord lheap ⇒ nat where
t_del_min Leaf = 1 |
t_del_min (Node l a n r) = t_merge l r

lemma t_merge_rank: t_merge l r ≤ rank l + rank r + 1
proof(induction l r rule: merge.induct)
  case 3 thus ?case by(simp)
qed simp_all

corollary t_merge_log: assumes ltree l ltree r
  shows t_merge l r ≤ log 2 (size1 l) + log 2 (size1 r) + 1
using le_log2_of_power[OF pow2_rank_size1[OF assms(1)]]
  le_log2_of_power[OF pow2_rank_size1[OF assms(2)]] t_merge_rank[of l r]
by linarith

corollary t_insert_log: ltree t ⟹ t_insert x t ≤ log 2 (size1 t) + 2
using t_merge_log[of Node Leaf x 1 Leaf t]
by(simp add: t_insert_def split: tree.split)


lemma ld_ld_1_less:
  assumes x > 0 y > 0 shows log 2 x + log 2 y + 1 < 2 * log 2 (x+y)
proof −
  have 2 powr (log 2 x + log 2 y + 1) = 2*x*y
    using assms by(simp add: powr_add)
  also have ... < (x+y)^2 using assms
    by(simp add: numeral_eq_Suc algebra_simps add_pos_pos)
  also have ... = 2 powr (2 * log 2 (x+y))
    using assms by(simp add: powr_add log_powr[symmetric])
  finally show ?thesis by simp
qed

corollary t_del_min_log: assumes ltree t
  shows t_del_min t ≤ 2 * log 2 (size1 t) + 1
proof(cases t)
  case Leaf thus ?thesis using assms by simp
next
  case [simp]: (Node t1 _ _ t2)
  have t_del_min t = t_merge t1 t2 by simp
  also have ... ≤ log 2 (size1 t1) + log 2 (size1 t2) + 1
    using ‹ltree t› by (auto simp: t_merge_log simp del: t_merge.simps)
  also have ... ≤ 2 * log 2 (size1 t) + 1
    using ld_ld_1_less[of size1 t1 size1 t2] by (simp)
  finally show ?thesis .
```

qed

end

38 Binomial Heap

theory *Binomial_Heap*
imports
Base_FDS
Complex_Main
Priority_Queue_Specs
begin

We formalize the binomial heap presentation from Okasaki's book. We show the functional correctness and complexity of all operations.

The presentation is engineered for simplicity, and most proofs are straightforward and automatic.

38.1 Binomial Tree and Heap Datatype

datatype *'a tree* = *Node* (*rank*: *nat*) (*root*: *'a*) (*children*: *'a tree list*)

type_synonym *'a heap* = *'a tree list*

38.1.1 Multiset of elements

fun *mset_tree* :: *'a*::*linorder tree* ⇒ *'a multiset* **where**
mset_tree (*Node* _ *a c*) = {#*a*#} + (∑ *t*∈#*mset c*. *mset_tree t*)

definition *mset_heap* :: *'a*::*linorder heap* ⇒ *'a multiset* **where**
mset_heap c = (∑ *t*∈#*mset c*. *mset_tree t*)

lemma *mset_tree_simp_alt*[*simp*]:
mset_tree (*Node r a c*) = {#*a*#} + *mset_heap c*
unfolding *mset_heap_def* **by** *auto*
declare *mset_tree.simps*[*simp del*]

lemma *mset_tree_nonempty*[*simp*]: *mset_tree t* ≠ {#}
by (*cases t*) *auto*

lemma *mset_heap_Nil*[*simp*]:
mset_heap [] = {#}
by (*auto simp*: *mset_heap_def*)

lemma *mset_heap_Cons*[*simp*]: *mset_heap* (*t*#*ts*) = *mset_tree t* + *mset_heap ts*
by (*auto simp*: *mset_heap_def*)

lemma *mset_heap_empty_iff*[*simp*]: *mset_heap ts* = {#} ⟷ *ts*=[]
by (*auto simp*: *mset_heap_def*)

lemma *root_in_mset*[*simp*]: *root t* ∈# *mset_tree t*
by (*cases t*) *auto*

lemma *mset_heap_rev_eq*[*simp*]: *mset_heap* (*rev ts*) = *mset_heap ts*
by (*auto simp*: *mset_heap_def*)

38.1.2 Invariants

Binomial invariant

fun *invar_btree* :: 'a::*linorder tree* ⇒ *bool* **where**
invar_btree (*Node r x ts*) ⟷
  (∀ *t*∈*set ts*. *invar_btree t*) ∧ *map rank ts* = *rev* [*0*..<*r*]

definition *invar_bheap* :: 'a::*linorder heap* ⇒ *bool* **where**
invar_bheap ts
  ⟷ (∀ *t*∈*set ts*. *invar_btree t*) ∧ (*sorted_wrt* (<) (*map rank ts*))

Ordering (heap) invariant

fun *invar_otree* :: 'a::*linorder tree* ⇒ *bool* **where**
invar_otree (*Node* _ *x ts*) ⟷ (∀ *t*∈*set ts*. *invar_otree t* ∧ *x* ≤ *root t*)

definition *invar_oheap* :: 'a::*linorder heap* ⇒ *bool* **where**
invar_oheap ts ⟷ (∀ *t*∈*set ts*. *invar_otree t*)

definition *invar* :: 'a::*linorder heap* ⇒ *bool* **where**
invar ts ⟷ *invar_bheap ts* ∧ *invar_oheap ts*

The children of a node are a valid heap

lemma *invar_oheap_children*:
  invar_otree (*Node r v ts*) ⟹ *invar_oheap* (*rev ts*)
by (*auto simp*: *invar_oheap_def*)

lemma *invar_bheap_children*:
  invar_btree (*Node r v ts*) ⟹ *invar_bheap* (*rev ts*)
by (*auto simp*: *invar_bheap_def rev_map*[*symmetric*])

38.2 Operations and Their Functional Correctness

38.2.1 *link*

context
includes *pattern_aliases*
begin

fun *link* :: ('*a*::*linorder*) *tree* ⇒ '*a tree* ⇒ '*a tree* **where**
  link (*Node r* x_1 ts_1 =: t_1) (*Node r'* x_2 ts_2 =: t_2) =
    (*if* $x_1 \leq x_2$ *then Node* (*r*+*1*) x_1 (t_2#ts_1) *else Node* (*r*+*1*) x_2 (t_1#ts_2))

end

lemma *invar_btree_link*:
  assumes *invar_btree* t_1
  assumes *invar_btree* t_2
  assumes *rank* t_1 = *rank* t_2
  shows *invar_btree* (*link* t_1 t_2)
using *assms*
by (*cases* (t_1, t_2) *rule*: *link.cases*) *simp*

lemma *invar_link_otree*:
  assumes *invar_otree* t_1
  assumes *invar_otree* t_2
  shows *invar_otree* (*link* t_1 t_2)
using *assms*
by (*cases* (t_1, t_2) *rule*: *link.cases*) *auto*

lemma *rank_link*[*simp*]: *rank* (*link* t_1 t_2) = *rank* t_1 + *1*
by (*cases* (t_1, t_2) *rule*: *link.cases*) *simp*

lemma *mset_link*[*simp*]: *mset_tree* (*link* t_1 t_2) = *mset_tree* t_1 + *mset_tree* t_2
by (*cases* (t_1, t_2) *rule*: *link.cases*) *simp*

38.2.2 *ins_tree*

fun *ins_tree* :: '*a*::*linorder tree* ⇒ '*a heap* ⇒ '*a heap* **where**
  ins_tree t [] = [*t*]
| *ins_tree* t_1 (t_2#*ts*) =
  (*if rank* t_1 < *rank* t_2 *then* t_1#t_2#*ts else ins_tree* (*link* t_1 t_2) *ts*)

lemma *invar_bheap_Cons*[*simp*]:
  invar_bheap (*t*#*ts*)

$\longleftrightarrow$ *invar_btree t* $\wedge$ *invar_bheap ts* $\wedge$ $(\forall t' \in set\ ts.\ rank\ t < rank\ t')$
by (*auto simp*: *invar_bheap_def*)

lemma *invar_btree_ins_tree*:
assumes *invar_btree t*
assumes *invar_bheap ts*
assumes $\forall t' \in set\ ts.\ rank\ t \leq rank\ t'$
shows *invar_bheap* (*ins_tree t ts*)
using *assms*
by (*induction t ts rule*: *ins_tree.induct*) (*auto simp*: *invar_btree_link less_eq_Suc_le*[*symmetric*])

lemma *invar_oheap_Cons*[*simp*]:
invar_oheap (*t*#*ts*) $\longleftrightarrow$ *invar_otree t* $\wedge$ *invar_oheap ts*
by (*auto simp*: *invar_oheap_def*)

lemma *invar_oheap_ins_tree*:
assumes *invar_otree t*
assumes *invar_oheap ts*
shows *invar_oheap* (*ins_tree t ts*)
using *assms*
by (*induction t ts rule*: *ins_tree.induct*) (*auto simp*: *invar_link_otree*)

lemma *mset_heap_ins_tree*[*simp*]:
mset_heap (*ins_tree t ts*) = *mset_tree t* + *mset_heap ts*
by (*induction t ts rule*: *ins_tree.induct*) *auto*

lemma *ins_tree_rank_bound*:
assumes $t' \in set$ (*ins_tree t ts*)
assumes $\forall t' \in set\ ts.\ rank\ t_0 < rank\ t'$
assumes $rank\ t_0 < rank\ t$
shows $rank\ t_0 < rank\ t'$
using *assms*
by (*induction t ts rule*: *ins_tree.induct*) (*auto split*: *if_splits*)

38.2.3 *insert*

hide_const (**open**) *insert*

definition *insert* :: *'a*::*linorder* $\Rightarrow$ *'a heap* $\Rightarrow$ *'a heap* **where**
insert x ts = *ins_tree* (*Node 0 x* []) *ts*

lemma *invar_insert*[*simp*]: *invar t* $\Longrightarrow$ *invar* (*insert x t*)
by (*auto intro*!: *invar_btree_ins_tree simp*: *invar_oheap_ins_tree insert_def invar_def*)

lemma *mset_heap_insert*[*simp*]: *mset_heap* (*insert x t*) = {#*x*#} + *mset_heap t*
by(*auto simp*: *insert_def*)

38.2.4 *merge*

fun *merge* :: *'a*::*linorder heap* ⇒ *'a heap* ⇒ *'a heap* **where**
  merge ts_1 [] = ts_1
| *merge* [] ts_2 = ts_2
| *merge* (t_1#ts_1) (t_2#ts_2) = (
    if rank t_1 < *rank* t_2 *then* t_1 # *merge* ts_1 (t_2#ts_2) *else*
    if rank t_2 < *rank* t_1 *then* t_2 # *merge* (t_1#ts_1) ts_2
    else ins_tree (*link* t_1 t_2) (*merge* ts_1 ts_2)
)

lemma *merge_simp2*[*simp*]: *merge* [] ts_2 = ts_2
by (*cases* ts_2) *auto*

lemma *merge_rank_bound*:
  assumes *t'* ∈ *set* (*merge* ts_1 ts_2)
  assumes ∀ *t'*∈*set* ts_1. *rank t* < *rank t'*
  assumes ∀ *t'*∈*set* ts_2. *rank t* < *rank t'*
  shows *rank t* < *rank t'*
using *assms*
by (*induction* ts_1 ts_2 *arbitrary*: *t'* *rule*: *merge.induct*)
   (*auto split*: *if_splits simp*: *ins_tree_rank_bound*)

lemma *invar_bheap_merge*:
  assumes *invar_bheap* ts_1
  assumes *invar_bheap* ts_2
  shows *invar_bheap* (*merge* ts_1 ts_2)
  using *assms*
proof (*induction* ts_1 ts_2 *rule*: *merge.induct*)
  case (*3* t_1 ts_1 t_2 ts_2)

  from *3.prems* **have** [*simp*]: *invar_btree* t_1 *invar_btree* t_2
    by *auto*

  consider (*LT*) *rank* t_1 < *rank* t_2
        | (*GT*) *rank* t_1 > *rank* t_2
        | (*EQ*) *rank* t_1 = *rank* t_2
    using *antisym_conv3* **by** *blast*
  then show *?case* **proof** *cases*

```
    case LT
    then show ?thesis using 3
      by (force elim!: merge_rank_bound)
  next
    case GT
    then show ?thesis using 3
      by (force elim!: merge_rank_bound)
  next
    case [simp]: EQ

    from 3.IH(3) 3.prems have [simp]: invar_bheap (merge ts₁ ts₂)
      by auto

    have rank t₂ < rank t' if t' ∈ set (merge ts₁ ts₂) for t'
      using that
      apply (rule merge_rank_bound)
      using 3.prems by auto
    with EQ show ?thesis
      by (auto simp: Suc_le_eq invar_btree_ins_tree invar_btree_link)
  qed
qed simp_all

lemma invar_oheap_merge:
  assumes invar_oheap ts₁
  assumes invar_oheap ts₂
  shows invar_oheap (merge ts₁ ts₂)
using assms
by (induction ts₁ ts₂ rule: merge.induct)
   (auto simp: invar_oheap_ins_tree invar_link_otree)

lemma invar_merge[simp]: ⟦ invar ts₁; invar ts₂ ⟧ ⟹ invar (merge ts₁
ts₂)
by (auto simp: invar_def invar_bheap_merge invar_oheap_merge)

lemma mset_heap_merge[simp]:
  mset_heap (merge ts₁ ts₂) = mset_heap ts₁ + mset_heap ts₂
by (induction ts₁ ts₂ rule: merge.induct) auto
```

38.2.5 *get_min*

```
fun get_min :: 'a::linorder heap ⇒ 'a where
  get_min [t] = root t
| get_min (t#ts) = min (root t) (get_min ts)
```

lemma *invar_otree_root_min*:
  assumes *invar_otree t*
  assumes *x* ∈# *mset_tree t*
  shows *root t* ≤ *x*
using *assms*
by (*induction t arbitrary*: *x rule*: *mset_tree.induct*) (*fastforce simp*: *mset_heap_def*)

lemma *get_min_mset_aux*:
  assumes *ts*≠[]
  assumes *invar_oheap ts*
  assumes *x* ∈# *mset_heap ts*
  shows *get_min ts* ≤ *x*
  using *assms*
apply (*induction ts arbitrary*: *x rule*: *get_min.induct*)
apply (*auto*
    simp: *invar_otree_root_min min_def intro*: *order_trans*;
    meson linear order_trans invar_otree_root_min
)+
done

lemma *get_min_mset*:
  assumes *ts*≠[]
  assumes *invar ts*
  assumes *x* ∈# *mset_heap ts*
  shows *get_min ts* ≤ *x*
using *assms* **by** (*auto simp*: *invar_def get_min_mset_aux*)

lemma *get_min_member*:
  ts≠[] ⟹ *get_min ts* ∈# *mset_heap ts*
by (*induction ts rule*: *get_min.induct*) (*auto simp*: *min_def*)

lemma *get_min*:
  assumes *mset_heap ts* ≠ {#}
  assumes *invar ts*
  shows *get_min ts* = *Min_mset* (*mset_heap ts*)
using *assms get_min_member get_min_mset*
by (*auto simp*: *eq_Min_iff*)

38.2.6 *get_min_rest*

fun *get_min_rest* :: *'a*::*linorder heap* ⇒ *'a tree* × *'a heap* **where**
  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'*))

lemma *get_min_rest_get_min_same_root*:
assumes *ts*≠[]
assumes *get_min_rest ts* = (*t'*,*ts'*)
shows *root t'* = *get_min ts*
using *assms*
by (*induction ts arbitrary*: *t' ts' rule*: *get_min.induct*) (*auto simp*: *min_def split*: *prod.splits*)

lemma *mset_get_min_rest*:
assumes *get_min_rest ts* = (*t'*,*ts'*)
assumes *ts*≠[]
shows *mset ts* = {#*t'*#} + *mset ts'*
using *assms*
by (*induction ts arbitrary*: *t' ts' rule*: *get_min.induct*) (*auto split*: *prod.splits if_splits*)

lemma *set_get_min_rest*:
assumes *get_min_rest ts* = (*t'*, *ts'*)
assumes *ts*≠[]
shows *set ts* = *Set.insert t'* (*set ts'*)
using *mset_get_min_rest*[*OF assms*, *THEN arg_cong*[**where** *f*=*set_mset*]]
by *auto*

lemma *invar_bheap_get_min_rest*:
assumes *get_min_rest ts* = (*t'*,*ts'*)
assumes *ts*≠[]
assumes *invar_bheap ts*
shows *invar_btree t'* **and** *invar_bheap ts'*
proof −
have *invar_btree t'* ∧ *invar_bheap ts'*
using *assms*
proof (*induction ts arbitrary*: *t' ts' rule*: *get_min.induct*)
case (*2 t v va*)
then show *?case*
apply (*clarsimp split*: *prod.splits if_splits*)
apply (*drule set_get_min_rest*; *fastforce*)
done
qed *auto*
thus *invar_btree t'* **and** *invar_bheap ts'* **by** *auto*
qed

lemma *invar_oheap_get_min_rest*:
assumes *get_min_rest ts* = (*t'*,*ts'*)

assumes *ts*≠[]
assumes *invar_oheap ts*
shows *invar_otree t′* **and** *invar_oheap ts′*
using *assms*
by (*induction ts arbitrary*: *t′ ts′ rule*: *get_min.induct*) (*auto split*: *prod.splits if_splits*)

38.2.7 *del_min*

definition *del_min* :: *′a*::*linorder heap* ⇒ *′a*::*linorder heap* **where**
del_min ts = (*case get_min_rest ts of*
(*Node r x ts*$_1$, *ts*$_2$) ⇒ *merge* (*rev ts*$_1$) *ts*$_2$)

lemma *invar_del_min*[*simp*]:
assumes *ts* ≠ []
assumes *invar ts*
shows *invar* (*del_min ts*)
using *assms*
unfolding *invar_def del_min_def*
by (*auto*
split: *prod.split tree.split*
intro!: *invar_bheap_merge invar_oheap_merge*
dest: *invar_bheap_get_min_rest invar_oheap_get_min_rest*
intro!: *invar_oheap_children invar_bheap_children*
)

lemma *mset_heap_del_min*:
assumes *ts* ≠ []
shows *mset_heap ts* = *mset_heap* (*del_min ts*) + {# *get_min ts* #}
using *assms*
unfolding *del_min_def*
apply (*clarsimp split*: *tree.split prod.split*)
apply (*frule* (*1*) *get_min_rest_get_min_same_root*)
apply (*frule* (*1*) *mset_get_min_rest*)
apply (*auto simp*: *mset_heap_def*)
done

38.2.8 Instantiating the Priority Queue Locale

Last step of functional correctness proof: combine all the above lemmas to show that binomial heaps satisfy the specification of priority queues with merge.

interpretation *binheap*: *Priority_Queue_Merge*
where *empty* = [] **and** *is_empty* = (=) [] **and** *insert* = *insert*

and *get_min* = *get_min* **and** *del_min* = *del_min* **and** *merge* = *merge*
and *invar* = *invar* **and** *mset* = *mset_heap*
proof (*unfold_locales*, *goal_cases*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *auto*
next
case *3* **thus** *?case* **by** *auto*
next
case (*4 q*)
thus *?case* **using** *mset_heap_del_min*[*of q*] *get_min*[*OF _* ‹*invar q*›]
by (*auto simp*: *union_single_eq_diff*)
next
case (*5 q*) **thus** *?case* **using** *get_min*[*of q*] **by** *auto*
next
case *6* **thus** *?case* **by** (*auto simp add*: *invar_def invar_bheap_def invar_oheap_def*)
next
case *7* **thus** *?case* **by** *simp*
next
case *8* **thus** *?case* **by** *simp*
next
case *9* **thus** *?case* **by** *simp*
next
case *10* **thus** *?case* **by** *simp*
qed

38.3 Complexity

The size of a binomial tree is determined by its rank

lemma *size_mset_btree*:
assumes *invar_btree t*
shows *size* (*mset_tree t*) = *2^rank t*
using *assms*
proof (*induction t*)
case (*Node r v ts*)
hence *IH*: *size* (*mset_tree t*) = *2^rank t* **if** *t* ∈ *set ts* **for** *t*
using *that* **by** *auto*

from *Node* **have** *COMPL*: *map rank ts* = *rev* [*0*..<*r*] **by** *auto*

have *size* (*mset_heap ts*) = (∑ *t*←*ts*. *size* (*mset_tree t*))
by (*induction ts*) *auto*
also have ... = (∑ *t*←*ts*. *2^rank t*) **using** *IH*

```
    by (auto cong: map_cong)
  also have ... = (∑ r←map rank ts. 2^r)
    by (induction ts) auto
  also have ... = (∑ i∈{0..<r}. 2^i)
    unfolding COMPL
    by (auto simp: rev_map[symmetric] interv_sum_list_conv_sum_set_nat)
  also have ... = 2^r - 1
    by (induction r) auto
  finally show ?case
    by (simp)
qed
```

The length of a binomial heap is bounded by the number of its elements

```
lemma size_mset_bheap:
  assumes invar_bheap ts
  shows 2^length ts ≤ size (mset_heap ts) + 1
proof -
  from ‹invar_bheap ts› have
    ASC: sorted_wrt (<) (map rank ts) and
    TINV: ∀ t∈set ts. invar_btree t
    unfolding invar_bheap_def by auto

  have (2::nat)^length ts = (∑ i∈{0..<length ts}. 2^i) + 1
    by (simp add: sum_power2)
  also have ... ≤ (∑ t←ts. 2^rank t) + 1
   using sorted_wrt_less_sum_mono_lowerbound[OF _ ASC, of (^) (2::nat)]
    using power_increasing[where a=2::nat]
    by (auto simp: o_def)
  also have ... = (∑ t←ts. size (mset_tree t)) + 1 using TINV
    by (auto cong: map_cong simp: size_mset_btree)
  also have ... = size (mset_heap ts) + 1
    unfolding mset_heap_def by (induction ts) auto
  finally show ?thesis .
qed
```

38.3.1 Timing Functions

We define timing functions for each operation, and provide estimations of their complexity.

```
definition t_link :: 'a::linorder tree ⇒ 'a tree ⇒ nat where
[simp]: t_link _ _ = 1

fun t_ins_tree :: 'a::linorder tree ⇒ 'a heap ⇒ nat where
  t_ins_tree t [] = 1
```

```
| t_ins_tree t1 (t2 # rest) = (
    (if rank t1 < rank t2 then 1
     else t_link t1 t2 + t_ins_tree (link t1 t2) rest)
  )

definition t_insert :: 'a::linorder ⇒ 'a heap ⇒ nat where
t_insert x ts = t_ins_tree (Node 0 x []) ts

lemma t_ins_tree_simple_bound: t_ins_tree t ts ≤ length ts + 1
by (induction t ts rule: t_ins_tree.induct) auto
```

38.3.2 *t_insert*

```
lemma t_insert_bound:
  assumes invar ts
  shows t_insert x ts ≤ log 2 (size (mset_heap ts) + 1) + 1
proof −

  have 1: t_insert x ts ≤ length ts + 1
    unfolding t_insert_def by (rule t_ins_tree_simple_bound)
  also have ... ≤ log 2 (2 ∗ (size (mset_heap ts) + 1))
  proof −
    from size_mset_bheap[of ts] assms
    have 2 ^ length ts ≤ size (mset_heap ts) + 1
      unfolding invar_def by auto
    hence 2 ^ (length ts + 1) ≤ 2 ∗ (size (mset_heap ts) + 1) by auto
    thus ?thesis using le_log2_of_power by blast
  qed
  finally show ?thesis
    by (simp only: log_mult of_nat_mult) auto
qed
```

38.3.3 *t_merge*

```
fun t_merge :: 'a::linorder heap ⇒ 'a heap ⇒ nat where
  t_merge ts1 [] = 1
| t_merge [] ts2 = 1
| t_merge (t1#ts1) (t2#ts2) = 1 + (
    if rank t1 < rank t2 then t_merge ts1 (t2#ts2)
    else if rank t2 < rank t1 then t_merge (t1#ts1) ts2
    else t_ins_tree (link t1 t2) (merge ts1 ts2) + t_merge ts1 ts2
  )
```

A crucial idea is to estimate the time in correlation with the result length, as each carry reduces the length of the result.

lemma *t_ins_tree_length*:
t_ins_tree t ts + length (ins_tree t ts) = 2 + length ts
by (*induction t ts rule*: *ins_tree.induct*) *auto*

lemma *t_merge_length*:
length (merge ts_1 ts_2*) + t_merge* ts_1 $ts_2 \leq 2 * (length\ ts_1 + length\ ts_2) + 1$
by (*induction* ts_1 ts_2 *rule*: *t_merge.induct*)
(*auto simp*: *t_ins_tree_length algebra_simps*)

Finally, we get the desired logarithmic bound

lemma *t_merge_bound_aux*:
fixes ts_1 ts_2
defines $n_1 \equiv size\ (mset\_heap\ ts_1)$
defines $n_2 \equiv size\ (mset\_heap\ ts_2)$
assumes *BINVARS*: *invar_bheap* ts_1 *invar_bheap* ts_2
shows *t_merge* ts_1 $ts_2 \leq 4 * log\ 2\ (n_1 + n_2 + 1) + 2$
proof −
define n **where** $n = n_1 + n_2$

from *t_merge_length*[*of* ts_1 ts_2]
have *t_merge* ts_1 $ts_2 \leq 2 * (length\ ts_1 + length\ ts_2) + 1$ **by** *auto*
hence $(2::nat)\hat{}$*t_merge* ts_1 $ts_2 \leq 2\hat{}(2 * (length\ ts_1 + length\ ts_2) + 1)$
by (*rule power_increasing*) *auto*
also have $\ldots = 2*(2\hat{}length\ ts_1)^2*(2\hat{}length\ ts_2)^2$
by (*auto simp*: *algebra_simps power_add power_mult*)
also note *BINVARS*(*1*)[*THEN size_mset_bheap*]
also note *BINVARS*(*2*)[*THEN size_mset_bheap*]
finally have $2\ \hat{}$ *t_merge* ts_1 $ts_2 \leq 2 * (n_1 + 1)^2 * (n_2 + 1)^2$
by (*auto simp*: *power2_nat_le_eq_le* n_1*_def* n_2*_def*)
from *le_log2_of_power*[*OF this*] **have** *t_merge* ts_1 $ts_2 \leq log\ 2\ \ldots$
by *simp*
also have $\ldots = log\ 2\ 2 + 2*log\ 2\ (n_1 + 1) + 2*log\ 2\ (n_2 + 1)$
by (*simp add*: *log_mult log_nat_power*)
also have $n_2 \leq n$ **by** (*auto simp*: *n_def*)
finally have *t_merge* ts_1 $ts_2 \leq log\ 2\ 2 + 2*log\ 2\ (n_1 + 1) + 2*log\ 2\ (n + 1)$
by *auto*
also have $n_1 \leq n$ **by** (*auto simp*: *n_def*)
finally have *t_merge* ts_1 $ts_2 \leq log\ 2\ 2 + 4*log\ 2\ (n + 1)$
by *auto*
also have $log\ 2\ 2 \leq 2$ **by** *auto*
finally have *t_merge* ts_1 $ts_2 \leq 4*log\ 2\ (n + 1) + 2$ **by** *auto*
thus *?thesis* **unfolding** *n_def* **by** (*auto simp*: *algebra_simps*)

qed

lemma *t_merge_bound*:
fixes ts_1 ts_2
defines $n_1 \equiv$ *size* (*mset_heap* ts_1)
defines $n_2 \equiv$ *size* (*mset_heap* ts_2)
assumes *invar* ts_1 *invar* ts_2
shows *t_merge* ts_1 $ts_2 \leq 4 * log\ 2\ (n_1 + n_2 + 1) + 2$
using *assms t_merge_bound_aux* **unfolding** *invar_def* **by** *blast*

38.3.4 *t_get_min*

fun *t_get_min* :: *'a::linorder heap* ⇒ *nat* **where**
t_get_min [*t*] = *1*
| *t_get_min* (*t*#*ts*) = *1* + *t_get_min ts*

lemma *t_get_min_estimate*: *ts*≠[] ⟹ *t_get_min ts* = *length ts*
by (*induction ts rule*: *t_get_min.induct*) *auto*

lemma *t_get_min_bound*:
assumes *invar ts*
assumes *ts*≠[]
shows *t_get_min ts* ≤ *log 2* (*size* (*mset_heap ts*) + *1*)
proof −
have *1*: *t_get_min ts* = *length ts* **using** *assms t_get_min_estimate* **by** *auto*
also have ... ≤ *log 2* (*size* (*mset_heap ts*) + *1*)
proof −
from *size_mset_bheap*[*of ts*] *assms* **have** *2* ^ *length ts* ≤ *size* (*mset_heap ts*) + *1*
unfolding *invar_def* **by** *auto*
thus *?thesis* **using** *le_log2_of_power* **by** *blast*
qed
finally show *?thesis* **by** *auto*
qed

38.3.5 *t_del_min*

fun *t_get_min_rest* :: *'a::linorder heap* ⇒ *nat* **where**
t_get_min_rest [*t*] = *1*
| *t_get_min_rest* (*t*#*ts*) = *1* + *t_get_min_rest ts*

lemma *t_get_min_rest_estimate*: *ts*≠[] ⟹ *t_get_min_rest ts* = *length ts*
by (*induction ts rule*: *t_get_min_rest.induct*) *auto*

lemma *t_get_min_rest_bound_aux*:
assumes *invar_bheap ts*
assumes *ts≠[]*
shows *t_get_min_rest ts ≤ log 2 (size (mset_heap ts) + 1)*
proof −
have *1*: *t_get_min_rest ts = length ts* **using** *assms t_get_min_rest_estimate* **by** *auto*
also have *... ≤ log 2 (size (mset_heap ts) + 1)*
proof −
from *size_mset_bheap[of ts] assms* **have** *2 ^ length ts ≤ size (mset_heap ts) + 1*
by *auto*
thus *?thesis* **using** *le_log2_of_power* **by** *blast*
qed
finally show *?thesis* **by** *auto*
qed

lemma *t_get_min_rest_bound*:
assumes *invar ts*
assumes *ts≠[]*
shows *t_get_min_rest ts ≤ log 2 (size (mset_heap ts) + 1)*
using *assms t_get_min_rest_bound_aux* **unfolding** *invar_def* **by** *blast*

Note that although the definition of function *rev* has quadratic complexity, it can and is implemented (via suitable code lemmas) as a linear time function. Thus the following definition is justified:

definition *t_rev xs = length xs + 1*

definition *t_del_min* :: *'a::linorder heap ⇒ nat* **where**
t_del_min ts = t_get_min_rest ts + (case get_min_rest ts of (Node _ x ts$_1$, *ts*$_2$)
⇒ t_rev ts$_1$ *+ t_merge (rev ts*$_1$*) ts*$_2$
)

lemma *t_rev_ts1_bound_aux*:
fixes *ts*
defines *n ≡ size (mset_heap ts)*
assumes *BINVAR*: *invar_bheap (rev ts)*
shows *t_rev ts ≤ 1 + log 2 (n+1)*
proof −
have *t_rev ts = length ts + 1* **by** (*auto simp*: *t_rev_def*)
hence *2^t_rev ts = 2∗2^length ts* **by** *auto*
also have *... ≤ 2∗n+2* **using** *size_mset_bheap[OF BINVAR]* **by** (*auto simp*: *n_def*)

```
  finally have 2 ^ t_rev ts ≤ 2 * n + 2 .
  from le_log2_of_power[OF this] have t_rev ts ≤ log 2 (2 * (n + 1))
    by auto
  also have ... = 1 + log 2 (n+1)
    by (simp only: of_nat_mult log_mult) auto
  finally show ?thesis by (auto simp: algebra_simps)
qed

lemma t_del_min_bound_aux:
  fixes ts
  defines n ≡ size (mset_heap ts)
  assumes BINVAR: invar_bheap ts
  assumes ts≠[]
  shows t_del_min ts ≤ 6 * log 2 (n+1) + 3
proof −
  obtain r x ts1 ts2 where GM: get_min_rest ts = (Node r x ts1, ts2)
    by (metis surj_pair tree.exhaust_sel)

  note BINVAR' = invar_bheap_get_min_rest[OF GM ‹ts≠[]› BINVAR]
 hence BINVAR1: invar_bheap (rev ts1) by (blast intro: invar_bheap_children)

  define n1 where n1 = size (mset_heap ts1)
  define n2 where n2 = size (mset_heap ts2)

  have t_rev_ts1_bound: t_rev ts1 ≤ 1 + log 2 (n+1)
  proof −
    note t_rev_ts1_bound_aux[OF BINVAR1, simplified, folded n1_def]
    also have n1 ≤ n
      unfolding n1_def n_def
      using mset_get_min_rest[OF GM ‹ts≠[]›]
      by (auto simp: mset_heap_def)
    finally show ?thesis by (auto simp: algebra_simps)
  qed

  have t_del_min ts = t_get_min_rest ts + t_rev ts1 + t_merge (rev ts1) ts2
    unfolding t_del_min_def by (simp add: GM)
  also have ... ≤ log 2 (n+1) + t_rev ts1 + t_merge (rev ts1) ts2
    using t_get_min_rest_bound_aux[OF assms(2−)] by (auto simp: n_def)
  also have ... ≤ 2*log 2 (n+1) + t_merge (rev ts1) ts2 + 1
    using t_rev_ts1_bound by auto
  also have ... ≤ 2*log 2 (n+1) + 4 * log 2 (n1 + n2 + 1) + 3
    using t_merge_bound_aux[OF ‹invar_bheap (rev ts1)› ‹invar_bheap ts2›]
    by (auto simp: n1_def n2_def algebra_simps)
  also have n1 + n2 ≤ n
```

unfolding n_1_def n_2_def n_def
using $mset\_get\_min\_rest[OF\ GM\ \langle ts{\neq}[]\rangle]$
by (*auto simp*: $mset\_heap\_def$)
finally have $t\_del\_min\ ts \leq 6 * log\ 2\ (n{+}1) + 3$
by *auto*
thus *?thesis* **by** (*simp add*: *algebra_simps*)
qed

lemma $t\_del\_min\_bound$:
fixes *ts*
defines $n \equiv size\ (mset\_heap\ ts)$
assumes *invar ts*
assumes $ts{\neq}[]$
shows $t\_del\_min\ ts \leq 6 * log\ 2\ (n{+}1) + 3$
using *assms* $t\_del\_min\_bound\_aux$ **unfolding** $invar\_def$ **by** *blast*

end

39 Bibliographic Notes

Red-black trees The insert function follows Okasaki [14], the delete function Kahrs [10, 11].

2-3 trees Equational definitions were given by Hoffmann and O'Donnell [8] (only insertion) and Reade [18]. Our formalisation is based on the teaching material by Turbak [21] .

1-2 brother trees They were invented by Ottmann and Six [15, 16]. The functional version is due to Hinze [7].

AA trees They were invented by Arne Anderson [3]. Our formalisation follows Ragde [17] but fixes a number of mistakes.

Splay trees They were invented by Sleator and Tarjan [20]. Our formalisation follows Schoenmakers [19].

Join-based BSTs They were invented by Adams [1, 2] and analyzed by Blelloch *et al.* [4].

Leftist heaps They were invented by Crane [6]. A first functional implementation is due to Núñez *et al.* [13].

References

[1] S. Adams. Implementing sets efficiently in a functional language. Technical Report CSTR 92-10, University of Southampton, Department of Electronics and Computer Science, 1992.

[2] S. Adams. Efficient sets - A balancing act. *J. Funct. Program.*, 3(4):553–561, 1993.

[3] A. Andersson. Balanced search trees made simple. In *Algorithms and Data Structures (WADS '93)*, volume 709 of *LNCS*, pages 60–71. Springer, 1993.

[4] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *SPAA*, pages 253–264. ACM, 2016.

[5] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4. Eindhoven University of Techology, 1983.

[6] C. A. Crane. *Linear Lists and Prorty Queues as Balanced Binary Trees.* PhD thesis, Computer Science Department, Stanford University, 1972.

[7] R. Hinze. Purely functional 1-2 brother trees. *J. Functional Programming*, 19(6):633–644, 2009.

[8] C. M. Hoffmann and M. J. O'Donnell. Programming with equations. *ACM Trans. Program. Lang. Syst.*, 4(1):83–112, 1982.

[9] 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.

[10] S. Kahrs. Red black trees. http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html.

[11] S. Kahrs. Red-black trees with types. *J. Functional Programming*, 11(4):425–432, 2001.

[12] T. Nipkow. Automatic functional correctness proofs for functional search trees. http://www.in.tum.de/~nipkow/pubs/trees.html, Feb. 2016.

[13] M. Núñez, P. Palao, and R. Pena. A second year course on data structures based on functional programming. In P. H. Hartel and M. J. Plasmeijer, editors, *Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 65–84. Springer, 1995.

[14] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[15] T. Ottmann and H.-W. Six. Eine neue Klasse von ausgeglichenen Binärbäumen. *Angewandte Informatik*, 18(9):395–400, 1976.

[16] T. Ottmann and D. Wood. 1-2 brother trees or AVL trees revisited. *Comput. J.*, 23(3):248–255, 1980.

[17] P. Ragde. Simple balanced binary search trees. In Caldwell, Hölzenspies, and Achten, editors, *Trends in Functional Programming in Education*, volume 170 of *EPTCS*, pages 78–87, 2014.

[18] C. Reade. Balanced trees with removals: An exercise in rewriting and proof. *Sci. Comput. Program.*, 18(2):181–204, 1992.

[19] B. Schoenmakers. A systematic analysis of splaying. *Information Processing Letters*, 45:41–50, 1993.

[20] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[21] F. Turbak. CS230 Handouts — Spring 2007, 2007. http://cs.wellesley.edu/~cs230/spring07/handouts.html.