Functional Data Structures

Tobias Nipkow

December 12, 2021

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 [13].

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*) = {#*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*) = {*x*} ∪ *set xs*
by(*simp add*: *mset_insort flip*: *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 ... ≤ (length xs + 1) ^ 2 + length xs + 1 + 1
    using T_insort_length[of x isort xs] by (simp add: length_isort)
  also have ... ≤ (length(x#xs) + 1) ^ 2
    by (simp add: power2_eq_square)
  finally show ?case .
qed
```

1.2 Merge Sort

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

fun msort :: 'a::linorder list ⇒ 'a list where
msort xs = (let n = length xs in
  if n ≤ 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 ≤ 1
    thus ?thesis by(simp add: msort.simps[of xs])
  next
    assume ¬ ?n ≤ 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 ≤ 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)
  show ?case
    by (auto simp: msort.simps [of xs] 1 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

```
abbreviation mset_mset :: 'a list list ⇒ 'a multiset where
mset_mset xss ≡ ∑# (image_mset mset (mset xss))
```

```
lemma mset_merge_adj:
  mset_mset (merge_adj xss) = mset_mset xss
```

by(*induction xss rule*: *merge_adj.induct*) (*auto simp*: *mset_merge*)

lemma *mset_merge_all*:
  mset (*merge_all xss*) = *mset_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 multiset.map_comp comp_def*)

lemma *sorted_merge_adj*:
  $\forall$ *xs* $\in$ *set xss*. *sorted xs* $\Longrightarrow$ $\forall$ *xs* $\in$ *set* (*merge_adj xss*). *sorted xs*
by(*induction xss rule*: *merge_adj.induct*) (*auto simp*: *sorted_merge*)

lemma *sorted_merge_all*:
  $\forall$ *xs* $\in$ *set xss*. *sorted xs* $\Longrightarrow$ *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* $\Rightarrow$ *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* $\Rightarrow$ *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* $\Rightarrow$ *nat* **where**
C_msort_bu xs = *C_merge_all* (*map* (λx. [*x*]) *xs*)

lemma *length_merge_adj*:
  $[\![$ *even*(*length xss*); $\forall$ *xs* $\in$ *set xss*. *length xs* = *m* $]\!]$
  $\Longrightarrow$ $\forall$ *xs* $\in$ *set* (*merge_adj xss*). *length xs* = *2*∗*m*
by(*induction xss rule*: *merge_adj.induct*) (*auto simp*: *length_merge*)

lemma *C_merge_adj*: $\forall$ *xs* $\in$ *set xss*. *length xs* = *m* $\Longrightarrow$ *C_merge_adj xss*
$\leq$ *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*: *algebra_simps*)
qed

lemma *C_merge_all*: $[\![\forall xs \in set\ xss.\ length\ xs = m;\ length\ xss = 2\hat{}k]\!]$
$\Longrightarrow C\_merge\_all\ xss \leq m * k * 2\hat{}k$
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 ∗: $\forall x \in set(merge\_adj\ ?xss).\ length\ x = 2{*}m$ **.**
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 ... $\leq m * 2\hat{}k$ + *C_merge_all ?xss2*
using *3.prems*(*2*) *C_merge_adj*[*OF 3.prems*(*1*)] **by** (*auto simp*: *algebra_simps*)
also have ... $\leq m * 2\hat{}k + (2{*}m) * k' * 2\hat{}k'$
using *3.IH*[*OF* ∗ ∗∗] **by** *simp*
also have ... $= m * k * 2\hat{}k$
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*) = {#*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*: $\forall x\in set\ xs.\ f\ a \le f\ x \Longrightarrow insort\_key\ f\ a\ xs = a\ \#\ xs$
by (*cases xs*) *auto*

lemma *filter_insort_key_neg*:
  $\neg\ P\ x \Longrightarrow filter\ P\ (insort\_key\ f\ x\ xs) = filter\ P\ xs$
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 $\ldots = insort\_key\ f\ a\ (filter\ (\lambda y.\ f\ y = k)\ (isort\_key\ f\ xs))$
      by (*simp add*: *True filter_insort_key_pos sorted_isort_key*)
    also have $\ldots = insort\_key\ f\ a\ (filter\ (\lambda y.\ f\ y = k)\ xs)$ **by** (*simp add*: *Cons.IH*)
    also have $\ldots = a\ \#\ (filter\ (\lambda y.\ f\ y = k)\ xs)$ **by**(*simp add*: *True insort_is_Cons*)
    also have $\ldots = filter\ (\lambda y.\ f\ y = k)\ (a\ \#\ xs)$ **by** (*simp add*: *True*)
    finally show *?thesis* **.**
  qed
qed

end

2 Creating Almost Complete 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]
declare Let_def[simp]

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)

lemma bal_inorder:
  ⟦ n ≤ length xs; bal n xs = (t,zs) ⟧
  ⟹ xs = inorder t @ zs ∧ size t = n
proof(induction n arbitrary: xs t zs rule: less_induct)
  case (less n) show ?case
  proof cases
   assume n = 0 thus ?thesis using less.prems by (simp add: bal_simps)
  next
```

assume [*arith*]: $n \neq 0$
let *?m* = *n div 2* **let** *?m'* = *n* − *1* − *?m*
from *less.prems(2)* **obtain** *l r ys* **where**
b1: *bal ?m xs* = (*l*,*ys*) **and**
b2: *bal ?m'* (*tl ys*) = (*r*,*zs*) **and**
t: *t* = ⟨*l*, *hd ys*, *r*⟩
by(*auto simp*: *bal_simps split*: *prod.splits*)
have *IH1*: *xs* = *inorder l* @ *ys* ∧ *size l* = *?m*
using *b1 less.prems(1)* **by**(*intro less.IH*) *auto*
have *IH2*: *tl ys* = *inorder r* @ *zs* ∧ *size r* = *?m'*
using *b2 IH1 less.prems(1)* **by**(*intro less.IH*) *auto*
show *?thesis* **using** *t IH1 IH2 less.prems(1) hd_Cons_tl[of ys]* **by** *fastforce*
qed
qed

corollary *inorder_bal_list[simp]*:
$n \le$ *length xs* ⟹ *inorder(bal_list n xs)* = *take n xs*
unfolding *bal_list_def*
by (*metis* (*mono_tags*) *prod.collapse[of bal n xs] append_eq_conv_conj bal_inorder length_inorder*)

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

corollary *inorder_bal_tree*:
$n \le$ *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*)

The length/size lemmas below do not require the precondition $n \le$ *length xs* (or $n \le$ *size t*) that they come with. They could take advantage of the fact that *bal xs n* yields a result even if *length xs* < *n*. In that case the result will contain one or more occurrences of *hd* []. However, this is counter-intuitive and does not reflect the execution in an eager functional language.

lemma *bal_length*: ⟦ $n \le$ *length xs*; *bal n xs* = (*t*,*zs*) ⟧ ⟹ *length zs* = *length xs* − *n*
using *bal_inorder* **by** *fastforce*

corollary *size_bal_list[simp]*: $n \le$ *length xs* ⟹ *size(bal_list n xs)* = *n*
unfolding *bal_list_def* **using** *bal_inorder prod.exhaust_sel* **by** *blast*

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

corollary size_bal_tree[simp]: n ≤ size t ⟹ 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:
  ⟦ n ≤ length xs; bal n xs = (t,zs) ⟧ ⟹ min_height t = nat(⌊log 2 (n +
1)⌋)
proof(induction n arbitrary: xs t zs rule: less_induct)
  case (less n)
  show ?case
  proof cases
   assume n = 0 thus ?thesis using less.prems(2) by (simp add: bal_simps)
  next
    assume [arith]: n ≠ 0
    let ?m = n div 2 let ?m' = n − 1 − ?m
    from less.prems obtain l r ys where
      b1: bal ?m xs = (l,ys) and
      b2: bal ?m' (tl ys) = (r,zs) and
      t: t = ⟨l, hd ys, r⟩
      by(auto simp: bal_simps split: prod.splits)
    let ?hl = nat (floor(log 2 (?m + 1)))
    let ?hr = nat (floor(log 2 (?m' + 1)))
    have IH1: min_height l = ?hl using less.IH[OF _ _ b1] less.prems(1)
by simp
    have IH2: min_height r = ?hr
      using less.prems(1) bal_length[OF _ b1] b2 by(intro less.IH) auto
    have (n+1) div 2 ≥ 1 by arith
    hence 0: log 2 ((n+1) div 2) ≥ 0 by simp
    have ?m' ≤ ?m by arith
    hence le: ?hr ≤ ?hl by(simp add: nat_mono floor_mono)
    have min_height t = min ?hl ?hr + 1 by (simp add: t IH1 IH2)
    also have ... = ?hr + 1 using le by (simp add: min_absorb2)
    also have ?m' + 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:
  ⟦ n ≤ length xs; bal n xs = (t,zs) ⟧ ⟹ height t = nat ⌈log 2 (n + 1)⌉
proof(induction n arbitrary: xs t zs rule: less_induct)
  case (less n) show ?case
  proof cases
    assume n = 0 thus ?thesis
      using less.prems by (simp add: bal_simps)
  next
    assume [arith]: n ≠ 0
    let ?m = n div 2 let ?m' = n − 1 − ?m
    from less.prems obtain l r ys where
      b1: bal ?m xs = (l,ys) and
      b2: bal ?m' (tl ys) = (r,zs) and
      t: t = ⟨l, hd ys, r⟩
      by(auto simp: bal_simps split: prod.splits)
    let ?hl = nat ⌈log 2 (?m + 1)⌉
    let ?hr = nat ⌈log 2 (?m' + 1)⌉
     have IH1: height l = ?hl using less.IH[OF _ _ b1] less.prems(1) by
simp
    have IH2: height r = ?hr
      using b2 bal_length[OF _ b1] less.prems(1) by(intro less.IH) auto
    have 0: log 2 (?m + 1) ≥ 0 by simp
    have ?m' ≤ ?m by arith
    hence le: ?hr ≤ ?hl
      by(simp add: nat_mono ceiling_mono del: nat_ceiling_le_eq)
    have height t = max ?hl ?hr + 1 by (simp add: t IH1 IH2)
    also have ... = ?hl + 1 using le by (simp add: max_absorb1)
    also have ... = nat ⌈log 2 (?m + 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 acomplete_bal:
  assumes n ≤ length xs bal n xs = (t,ys) shows acomplete t
unfolding acomplete_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 ≤ size t ⟹ height (bal_tree n t) = nat⌈log 2 (n + 1)⌉
unfolding bal_list_def bal_tree_def
by (metis bal_list_def height_bal_list length_inorder)

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 acomplete_bal_list[simp]: n ≤ length xs ⟹ acomplete (bal_list
n xs)
unfolding bal_list_def by (metis  acomplete_bal prod.collapse)

corollary acomplete_balance_list[simp]: acomplete (balance_list xs)
by (simp add: balance_list_def)

corollary acomplete_bal_tree[simp]: n ≤ size t ⟹ acomplete (bal_tree n
t)
by (simp add: bal_tree_def)

corollary acomplete_balance_tree[simp]: acomplete (balance_tree t)
by (simp add: balance_tree_def)

lemma wbalanced_bal: ⟦ n ≤ length xs; bal n xs = (t,ys) ⟧ ⟹ wbalanced t
proof(induction n arbitrary: xs t ys rule: less_induct)
  case (less n)
  show ?case
  proof cases
    assume n = 0
    thus ?thesis using less.prems(2) by(simp add: bal_simps)
  next
    assume [arith]: n ≠ 0
    with less.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 split: prod.splits)
  have l: wbalanced l using less.IH[OF _ _ rec1] less.prems(1) by linarith
   have wbalanced r
    using rec1 rec2 bal_length[OF _ rec1] less.prems(1) by(intro less.IH)
auto
    with l t bal_length[OF _ rec1] less.prems(1) bal_inorder[OF _ rec1]
bal_inorder[OF _ rec2]
   show ?thesis by auto
  qed
qed
```

An alternative proof via *wbalanced ?t* $\Longrightarrow$ *acomplete ?t*:

```
lemma ⟦ n ≤ length xs; bal n xs = (t,ys) ⟧ ⟹ acomplete t
by(rule acomplete_if_wbalanced[OF wbalanced_bal])

lemma wbalanced_bal_list[simp]: n ≤ length xs ⟹ 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]: n ≤ size t ⟹ 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 $EQ[simp]$: $cmp\ x\ y = EQ \longleftrightarrow x = y$
and $GT[simp]$: $cmp\ x\ y = GT \longleftrightarrow x > y$
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*:

$sorted(xs\ @\ y\ \#\ ys) = (sorted(xs\ @\ [y]) \wedge sorted(y\ \#\ ys))$
by(*fastforce simp add*: *sorted_wrt_Cons sorted_wrt_append*)

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

lemma *sorted_mid_iff'*: $NO\_MATCH\ []\ ys \Longrightarrow$
$sorted(xs\ @\ y\ \#\ ys) = (sorted(xs\ @\ [y]) \wedge sorted(y\ \#\ ys))$
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*:
$ASSUMPTION(sorted(xs\ @\ [x])) \Longrightarrow x \leq y \Longrightarrow sorted\ (xs\ @\ [y])$
by (*auto simp add*: *sorted_wrt_append ASSUMPTION_def*)

lemma *sorted_Cons_le*:
$ASSUMPTION(sorted(x\ \#\ xs)) \Longrightarrow y \leq x \Longrightarrow sorted\ (y\ \#\ xs)$
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*:
$sorted(x\ \#\ xs) = ((\forall\, y \in set\ xs.\ x < y) \wedge sorted\ xs)$
by(*simp add*: *sorted_wrt_Cons*)

lemma *sorted_snoc_iff*:
$sorted(xs\ @\ [x]) = (sorted\ xs \wedge (\forall\, y \in set\ xs.\ y < x))$
by(*simp add*: *sorted_wrt_append*)

lemmas *isin_simps* = *sorted_mid_iff' 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*) = *set xs* ∪ {*x*}
by(*induction xs*) *auto*

lemma *sorted_ins_list*: *sorted xs* ⟹ *sorted*(*ins_list x xs*)
by(*induction xs rule*: *induct_list012*) *auto*

lemma *ins_list_sorted*: *sorted* (*xs* @ [*a*]) ⟹
  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*]) ⟹ *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*]) ⟹ *a* ≤ *x* ⟹
  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*]) ⟹ *x* < *a* ⟹
  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*) ⟹ *ins_list x xs* = *x* # *xs*
by (*induction xs*) *auto*

lemma *ins_list_snoc*: *sorted* (*xs* @ [*x*]) ⟹ *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 ⇒ 'a list ⇒ '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* ∉ *set xs* ⟹ *del_list x xs* = *xs*

by (*induct xs*) *simp_all*

lemma *set_del_list*:
  sorted xs $\Longrightarrow$ *set* (*del_list x xs*) = *set xs* $-$ {*x*}
by(*induct xs*) (*auto simp*: *sorted_Cons_iff*)

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* $\le$ *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 s* $\cup \{x\}$
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 ⇒ 't ⇒ 't
fixes delete :: 'a ⇒ 't ⇒ 't
fixes isin :: 't ⇒ 'a ⇒ bool
fixes inorder :: 't ⇒ 'a list
fixes inv :: 't ⇒ bool
assumes inorder_empty: inorder empty = []
assumes isin: inv t ∧ sorted(inorder t) ⟹
  isin t x = (x ∈ set (inorder t))
assumes inorder_insert: inv t ∧ sorted(inorder t) ⟹
  inorder(insert x t) = ins_list x (inorder t)
assumes inorder_delete: inv t ∧ sorted(inorder t) ⟹
  inorder(delete x t) = del_list x (inorder t)
assumes inorder_inv_empty:  inv empty
assumes inorder_inv_insert: inv t ∧ sorted(inorder t) ⟹ inv(insert x t)
assumes inorder_inv_delete: inv t ∧ sorted(inorder t) ⟹ inv(delete x t)

begin
```

It implements the traditional specification:

```
definition set :: 't ⇒ 'a set where
set = List.set o inorder

definition invar :: 't ⇒ bool where
invar t = (inv t ∧ 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 in-
var_def)
next
  case (4 s x) thus ?case
    by (auto simp: inorder_delete set_del_list invar_def set_def)
next
  case 5 thus ?case by(simp add: inorder_empty inorder_inv_empty in-
var_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))
```

Deletion by replacing:

```
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')
```

Deletion by joining:

```
fun join :: ('a::linorder)tree ⇒ 'a tree ⇒ 'a tree where
join t Leaf = t |
join Leaf t = t |
join (Node t1 a t2) (Node t3 b t4) =
  (case join t2 t3 of
     Leaf ⇒ Node t1 a (Node Leaf b t4) |
     Node u2 x u3 ⇒ Node (Node t1 a u2) x (Node u3 b t4))

fun delete2 :: 'a::linorder ⇒ 'a tree ⇒ 'a tree where
delete2 x Leaf = Leaf |
delete2 x (Node l a r) =
  (case cmp x a of
     LT ⇒  Node (delete2 x l) a r |
     GT ⇒  Node l a (delete2 x r) |
     EQ ⇒ join l r)
```

7.1 Functional Correctness Proofs

lemma *isin_set*: *sorted*(*inorder t*) $\Longrightarrow$ *isin t x* = (*x* ∈ *set* (*inorder t*))

by (*induction t*) (*auto simp*: *isin_simps*)

lemma *inorder_insert*:
  sorted(*inorder t*) $\Longrightarrow$ *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′*) $\Longrightarrow$ *t* ≠ *Leaf* $\Longrightarrow$ *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*) $\Longrightarrow$ *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*)+

lemma *inorder_join*:
  inorder(*join l r*) = *inorder l* @ *inorder r*
by(*induction l r rule*: *join.induct*) (*auto split*: *tree.split*)

lemma *inorder_delete2*:
  sorted(*inorder t*) $\Longrightarrow$ *inorder*(*delete2 x t*) = *del_list x* (*inorder t*)
by(*induction t*) (*auto simp*: *inorder_join del_list_simps*)

interpretation *S2*: *Set_by_Ordered*
where *empty* = *empty* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete2*
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_delete2*)
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 [] = (*λx. None*) |
map_of ((*a*,*b*)#*ps*) = (*λx. 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*) $\Longrightarrow$ *map_of ps x* = *None*
by (*induction ps*) (*fastforce simp*: *sorted_lems sorted_wrt_Cons*)+

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

lemma *map_of_del_list*: *sorted1 ps* $\Longrightarrow$
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*) $\Longrightarrow$ *x* < *a* $\Longrightarrow$
map_of ps x = *None*
by (*simp add*: *map_of_None sorted_Cons_le*)

lemma *map_of_sorted_snoc*: *sorted* (*map fst ps* @ [*a*]) $\Longrightarrow$ *a* ≤ *x* $\Longrightarrow$
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* $\Longrightarrow$ *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*)]) $\Longrightarrow$
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*)]) $\Longrightarrow$ *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*) $\Longrightarrow$ *x* < *a* $\Longrightarrow$
  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*) $\Longrightarrow$ *x* < *b* $\Longrightarrow$
  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*) $\Longrightarrow$ *x* < *c* $\Longrightarrow$
  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*) $\Longrightarrow$ *x* < *d* $\Longrightarrow$
   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*) $\Longrightarrow$ *del_list x xs* = *xs*
by(*induction xs*)(*fastforce simp*: *sorted_wrt_Cons*)+

lemma *del_list_sorted_app*:
  sorted(*map fst xs* @ [*x*]) $\Longrightarrow$ *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* = (λ_. *None*)
and *map_update*: *invar m* ⟹ *lookup*(*update a b m*) = (*lookup m*)(*a* := *Some b*)
and *map_delete*: *invar m* ⟹ *lookup*(*delete a m*) = (*lookup m*)(*a* := *None*)
and *invar_empty*: *invar empty*
and *invar_update*: *invar m* ⟹ *invar*(*update a b m*)
and *invar_delete*: *invar m* ⟹ *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* ∧ *sorted1* (*inorder t*) ⟹
  lookup t a = *map_of* (*inorder t*) *a*
and *inorder_update*: *inv t* ∧ *sorted1* (*inorder t*) ⟹
  inorder(*update a b t*) = *upd_list a b* (*inorder t*)
and *inorder_delete*: *inv t* ∧ *sorted1* (*inorder t*) ⟹
  inorder(*delete a t*) = *del_list a* (*inorder t*)
and *inorder_inv_empty*: *inv empty*
and *inorder_inv_update*: *inv t* ∧ *sorted1* (*inorder t*) ⟹ *inv*(*update a b t*)
and *inorder_inv_delete*: *inv t* ∧ *sorted1* (*inorder t*) ⟹ *inv*(*delete a t*)

begin

It implements the traditional specification:

```
definition invar :: 't ⇒ bool where
invar t == inv t ∧ 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 ⇒ '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)

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

11 Augmented Tree (Tree2)

theory *Tree2*
imports *HOL−Library.Tree*
begin

This theory provides the basic infrastructure for the type $('a \times 'b)$ *tree* of augmented trees where $'a$ is the key and $'b$ some additional information.

IMPORTANT: Inductions and cases analyses on augmented trees need to use the following two rules explicitly. They generate nodes of the form $\langle l, (a, b), r\rangle$ rather than $\langle l, a, r\rangle$ for trees of type $'a$ *tree*.

lemmas *tree2_induct* = *tree.induct*[**where** $'a = 'a * 'b$, *split_format*(*complete*)]

lemmas *tree2_cases* = *tree.exhaust*[**where** $'a = 'a * 'b$, *split_format*(*complete*)]

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

fun *set_tree* :: $('a*'b)$ *tree* ⇒ $'a$ *set* **where**
set_tree Leaf = {} |
set_tree (*Node l* (*a*,_) *r*) = {*a*} ∪ *set_tree l* ∪ *set_tree r*

fun *bst* :: $('a::linorder*'b)$ *tree* ⇒ *bool* **where**
bst Leaf = *True* |
bst (*Node l* (*a*, _) *r*) = ((∀ *x* ∈ *set_tree l*. *x* < *a*) ∧ (∀ *x* ∈ *set_tree r*. *a* < *x*) ∧ *bst l* ∧ *bst r*)

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

lemma *eq_set_tree_empty*[*simp*]: *set_tree t* = {} ⟷ *t* = *Leaf*
by (*cases t*) *auto*

lemma *set_inorder*[*simp*]: *set* (*inorder t*) = *set_tree t*
by (*induction t*) *auto*

lemma *length_inorder*[*simp*]: *length* (*inorder t*) = *size t*
by (*induction t*) *auto*

end

12 Function *isin* for Tree2

theory *Isin2*
imports
  Tree2
  Cmp
  Set_Specs
begin

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

lemma *isin_set_inorder*: *sorted*(*inorder t*) $\Longrightarrow$ *isin t x* = ($x \in$ *set*(*inorder t*))
by (*induction t rule*: *tree2_induct*) (*auto simp*: *isin_simps*)

lemma *isin_set_tree*: *bst t* $\Longrightarrow$ *isin t x* $\longleftrightarrow$ $x \in$ *set_tree t*
by(*induction t rule*: *tree2_induct*) *auto*

end

13 Interval Trees

theory *Interval_Tree*
imports
  HOL−Data_Structures.Cmp
  HOL−Data_Structures.List_Ins_Del
  HOL−Data_Structures.Isin2
  HOL−Data_Structures.Set_Specs
begin

13.1 Intervals

The following definition of intervals uses the **typedef** command to define the type of non-empty intervals as a subset of the type of pairs *p* where *fst* *p* ≤ *snd p*:

```
typedef (overloaded) 'a::linorder ivl =
  {p :: 'a × 'a. fst p ≤ snd p} by auto
```

More precisely, *'a ivl* is isomorphic with that subset via the function *Rep_ivl*. Hence the basic interval properties are not immediate but need simple proofs:

```
definition low :: 'a::linorder ivl ⇒ 'a where
low p = fst (Rep_ivl p)

definition high :: 'a::linorder ivl ⇒ 'a where
high p = snd (Rep_ivl p)

lemma ivl_is_interval: low p ≤ high p
by (metis Rep_ivl high_def low_def mem_Collect_eq)

lemma ivl_inj: low p = low q ⟹ high p = high q ⟹ p = q
by (metis Rep_ivl_inverse high_def low_def prod_eqI)
```

Now we can forget how exactly intervals were defined.

```
instantiation ivl :: (linorder) linorder begin

definition ivl_less: (x < y) = (low x < low y | (low x = low y ∧ high x <
high y))
definition ivl_less_eq: (x ≤ y) = (low x < low y | (low x = low y ∧ high
x ≤ high y))

instance proof
  fix x y z :: 'a ivl
  show a: (x < y) = (x ≤ y ∧ ¬ y ≤ x)
    using ivl_less ivl_less_eq by force
  show b: x ≤ x
    by (simp add: ivl_less_eq)
  show c: x ≤ y ⟹ y ≤ z ⟹ x ≤ z
    using ivl_less_eq by fastforce
  show d: x ≤ y ⟹ y ≤ x ⟹ x = y
    using ivl_less_eq a ivl_inj ivl_less by fastforce
  show e: x ≤ y ∨ y ≤ x
    by (meson ivl_less_eq leI not_less_iff_gr_or_eq)
qed end


definition overlap :: ('a::linorder) ivl ⇒ 'a ivl ⇒ bool where
overlap x y ⟷ (high x ≥ low y ∧ high y ≥ low x)
```

definition *has_overlap* :: ($'a$::*linorder*) *ivl set* ⇒ $'a$ *ivl* ⇒ *bool* **where**
has_overlap S y ⟷ (∃ *x*∈*S*. *overlap x y*)

13.2 Interval Trees

type_synonym $'a$ *ivl_tree* = ($'a$ *ivl* ∗ $'a$) *tree*

fun *max_hi* :: ($'a$::*order_bot*) *ivl_tree* ⇒ $'a$ **where**
max_hi Leaf = *bot* |
max_hi (*Node* _ (_,*m*) _) = *m*

definition *max3* :: ($'a$::*linorder*) *ivl* ⇒ $'a$ ⇒ $'a$ ⇒ $'a$ **where**
max3 a m n = *max* (*high a*) (*max m n*)

fun *inv_max_hi* :: ($'a$::{*linorder*,*order_bot*}) *ivl_tree* ⇒ *bool* **where**
inv_max_hi Leaf ⟷ *True* |
inv_max_hi (*Node l* (*a*, *m*) *r*) ⟷ (*m* = *max3 a* (*max_hi l*) (*max_hi r*)
∧ *inv_max_hi l* ∧ *inv_max_hi r*)

```
lemma max_hi_is_max:
  inv_max_hi t ⟹ a ∈ set_tree t ⟹ high a ≤ max_hi t
by (induct t, auto simp add: max3_def max_def)

lemma max_hi_exists:
  inv_max_hi t ⟹ t ≠ Leaf ⟹ ∃ a∈set_tree t. high a = max_hi t
proof (induction t rule: tree2_induct)
  case Leaf
  then show ?case by auto
next
  case N: (Node l v m r)
  then show ?case
  proof (cases l rule: tree2_cases)
    case Leaf
    then show ?thesis
       using N.prems(1) N.IH(2) by (cases r, auto simp add: max3_def
max_def le_bot)
  next
    case Nl: Node
    then show ?thesis
    proof (cases r rule: tree2_cases)
      case Leaf
      then show ?thesis
      using N.prems(1) N.IH(1) Nl by (auto simp add: max3_def max_def
le_bot)
    next
      case Nr: Node
      obtain p1 where p1: p1 ∈ set_tree l high p1 = max_hi l
        using N.IH(1) N.prems(1) Nl by auto
      obtain p2 where p2: p2 ∈ set_tree r high p2 = max_hi r
        using N.IH(2) N.prems(1) Nr by auto
      then show ?thesis
        using p1 p2 N.prems(1) by (auto simp add: max3_def max_def)
    qed
  qed
qed
```

13.3 Insertion and Deletion

```
definition node where
[simp]: node l a r = Node l (a, max3 a (max_hi l) (max_hi r)) r

fun insert :: 'a::{linorder,order_bot} ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree where
insert x Leaf = Node Leaf (x, high x) Leaf |
insert x (Node l (a, m) r) =
  (case cmp x a of
     EQ ⇒ Node l (a, m) r |
     LT ⇒ node (insert x l) a r |
     GT ⇒ node l a (insert x r))

lemma inorder_insert:
  sorted (inorder t) ⟹ inorder (insert x t) = ins_list x (inorder t)
by (induct t rule: tree2_induct) (auto simp: ins_list_simps)

lemma inv_max_hi_insert:
  inv_max_hi t ⟹ inv_max_hi (insert x t)
by (induct t rule: tree2_induct) (auto simp add: max3_def)

fun split_min :: 'a::{linorder,order_bot} ivl_tree ⇒ 'a ivl × 'a ivl_tree
where
split_min (Node l (a, m) r) =
  (if l = Leaf then (a, r)
   else let (x,l') = split_min l in (x, node l' a r))

fun delete :: 'a::{linorder,order_bot} ivl ⇒ 'a ivl_tree ⇒ 'a ivl_tree where
delete x Leaf = Leaf |
delete x (Node l (a, m) 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')

lemma split_minD:
  split_min t = (x,t') ⟹ t ≠ Leaf ⟹ x # inorder t' = inorder t
by (induct 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 (induct t)
   (auto simp: del_list_simps split_minD Let_def split: prod.splits)

lemma inv_max_hi_split_min:
  ⟦ t ≠ Leaf;  inv_max_hi t ⟧ ⟹ inv_max_hi (snd (split_min t))
by (induct t) (auto split: prod.splits)

lemma inv_max_hi_delete:
  inv_max_hi t ⟹ inv_max_hi (delete x t)
apply (induct t)
 apply simp
using inv_max_hi_split_min by (fastforce simp add: Let_def split: prod.splits)
```

13.4 Search

Does interval x overlap with any interval in the tree?

```
fun search :: 'a::{linorder,order_bot} ivl_tree ⇒ 'a ivl ⇒ bool where
search Leaf x = False |
search (Node l (a, m) r) x =
  (if overlap x a then True
   else if l ≠ Leaf ∧ max_hi l ≥ low x then search l x
   else search r x)

lemma search_correct:
 inv_max_hi t ⟹ sorted (inorder t) ⟹ search t x = has_overlap (set_tree
t) x
proof (induction t rule: tree2_induct)
  case Leaf
  then show ?case by (auto simp add: has_overlap_def)
next
  case (Node l a m r)
  have search_l: search l x = has_overlap (set_tree l) x
```

```
    using Node.IH(1) Node.prems by (auto simp: sorted_wrt_append)
  have search_r: search r x = has_overlap (set_tree r) x
    using Node.IH(2) Node.prems by (auto simp: sorted_wrt_append)
  show ?case
  proof (cases overlap a x)
    case True
    thus ?thesis by (auto simp: overlap_def has_overlap_def)
  next
    case a_disjoint: False
    then show ?thesis
    proof cases
      assume [simp]: l = Leaf
      have search_eval: search (Node l (a, m) r) x = search r x
        using a_disjoint overlap_def by auto
      show ?thesis
        unfolding search_eval search_r
        by (auto simp add: has_overlap_def a_disjoint)
    next
      assume l ≠ Leaf
      then show ?thesis
      proof (cases max_hi l ≥ low x)
        case max_hi_l_ge: True
        have inv_max_hi l
          using Node.prems(1) by auto
        then obtain p where p: p ∈ set_tree l high p = max_hi l
          using ‹l ≠ Leaf› max_hi_exists by auto
        have search_eval: search (Node l (a, m) r) x = search l x
           using a_disjoint ‹l ≠ Leaf› max_hi_l_ge by (auto simp: over-
lap_def)
        show ?thesis
        proof (cases low p ≤ high x)
          case True
          have overlap p x
            unfolding overlap_def using True p(2) max_hi_l_ge by auto
          then show ?thesis
            unfolding search_eval search_l
            using p(1) by(auto simp: has_overlap_def overlap_def)
        next
          case False
          have ¬overlap x rp if asm: rp ∈ set_tree r for rp
          proof −
            have low p ≤ low rp
             using asm p(1) Node(4) by(fastforce simp: sorted_wrt_append
ivl_less)
```

```
          then show ?thesis
            using False by (auto simp: overlap_def)
        qed
        then show ?thesis
          unfolding search_eval search_l
          using a_disjoint by (auto simp: has_overlap_def overlap_def)
      qed
    next
      case False
      have search_eval: search (Node l (a, m) r) x = search r x
        using a_disjoint False by (auto simp: overlap_def)
      have ¬overlap x lp if asm: lp ∈ set_tree l for lp
        using asm False Node.prems(1) max_hi_is_max
        by (fastforce simp: overlap_def)
      then show ?thesis
        unfolding search_eval search_r
        using a_disjoint by (auto simp: has_overlap_def overlap_def)
    qed
  qed
 qed
qed

definition empty :: 'a ivl_tree where
empty = Leaf
```

13.5 Specification

```
locale Interval_Set = Set +
  fixes has_overlap :: 't ⇒ 'a::linorder ivl ⇒ bool
  assumes set_overlap: invar s ⟹ has_overlap s x = Interval_Tree.has_overlap
(set s) x

fun invar :: ('a::{linorder,order_bot}) ivl_tree ⇒ bool where
invar t = (inv_max_hi t ∧ sorted(inorder t))

interpretation S: Interval_Set
  where empty = Leaf and insert = insert and delete = delete
  and has_overlap = search and isin = isin and set = set_tree
  and invar = invar
proof (standard, goal_cases)
  case 1
  then show ?case by auto
next
  case 2
  then show ?case by (simp add: isin_set_inorder)
next
  case 3
  then show ?case by(simp add: inorder_insert set_ins_list flip: set_inorder)
next
  case 4
  then show ?case by(simp add: inorder_delete set_del_list flip: set_inorder)
next
  case 5
  then show ?case by auto
next
  case 6
  then show ?case by (simp add: inorder_insert inv_max_hi_insert sorted_ins_list)
next
  case 7
  then show ?case by (simp add: inorder_delete inv_max_hi_delete sorted_del_list)
next
  case 8
  then show ?case by (simp add: search_correct)
qed

end
```

14 AVL Tree Implementation of Sets

```
theory AVL_Set_Code
imports
  Cmp
  Isin2
begin
```

14.1 Code

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

definition empty :: 'a tree_ht where
empty = Leaf

fun ht :: 'a tree_ht ⇒ nat where
ht Leaf = 0 |
ht (Node l (a,n) r) = n

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

```
definition balL :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht where
balL AB c C =
  (if ht AB = ht C + 2 then
     case AB of
       Node A (a, _) B ⇒
         if ht A ≥ ht B then node A a (node B c C)
         else
           case B of
             Node B1 (b, _) B2 ⇒ node (node A a B1) b (node B2 c C)
   else node AB c C)

definition balR :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht where
balR A a BC =
   (if ht BC = ht A + 2 then
      case BC of
        Node B (c, _) C ⇒
          if ht B ≤ ht C then node (node A a B) c C
          else
            case B of
              Node B1 (b, _) B2 ⇒ node (node A a B1) b (node B2 c C)
  else node A a BC)

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

fun split_max :: 'a tree_ht ⇒ 'a tree_ht * '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 delete :: 'a::linorder ⇒ 'a tree_ht ⇒ 'a tree_ht where
delete _ Leaf = Leaf |
delete x (Node l (a, n) r) =
  (case cmp x a of
     EQ ⇒ if l = Leaf then r
           else let (l', a') = split_max l in balR l' a' r |
     LT ⇒ balR (delete x l) a r |
     GT ⇒ balL l a (delete x r))
```

14.2 Functional Correctness Proofs

Very different from the AFP/AVL proofs

14.2.1 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*)

14.2.2 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*)

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_split_maxD*
split: *prod.splits*)

end

14.3 Invariant

theory *AVL_Set*
imports
  AVL_Set_Code
  HOL−Number_Theory.Fib
begin

fun *avl* :: *′a tree_ht* ⇒ *bool* **where**
avl Leaf = *True* |

avl (*Node l* (*a*,*n*) *r*) =
 (*abs*(*int*(*height l*) − *int*(*height r*)) ≤ *1* ∧
  n = *max* (*height l*) (*height r*) + *1* ∧ *avl l* ∧ *avl r*)

14.3.1 Insertion maintains AVL balance

declare *Let_def* [*simp*]

lemma *ht_height*[*simp*]: *avl t* ⟹ *ht t* = *height t*
by (*cases t rule*: *tree2_cases*) *simp_all*

First, a fast but relatively manual proof with many lemmas:

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

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

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

lemma *avl_balL*:
  ⟦ *avl l*; *avl r*; *height r* − *1* ≤ *height l* ∧ *height l* ≤ *height r* + *2* ⟧ ⟹ *avl*(*balL l a r*)
by(*auto simp*: *balL_def node_def split*!: *if_split tree.split*)

lemma *avl_balR*:
  ⟦ *avl l*; *avl r*; *height l* − *1* ≤ *height r* ∧ *height r* ≤ *height l* + *2* ⟧ ⟹ *avl*(*balR l a r*)

by(*auto simp*: *balR_def node_def split*!: *if_split tree.split*)

Insertion maintains the AVL property. Requires simultaneous proof.

```
theorem avl_insert:
  avl t ⟹ avl(insert x t)
  avl t ⟹ height (insert x t) ∈ {height t, height t + 1}
proof (induction t rule: tree2_induct)
  case (Node l a _ r)
  case 1
  show ?case
  proof(cases x = a)
    case True with 1 show ?thesis by simp
  next
    case False
    show ?thesis
    proof(cases x<a)
      case True with 1 Node(1,2) show ?thesis by (auto intro!:avl_balL)
    next
      case False with 1 Node(3,4) ‹x≠a› show ?thesis by (auto intro!:avl_balR)
    qed
  qed
  case 2
  show ?case
  proof(cases x = a)
    case True with 2 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 2 Node(1,2) ‹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 2 Node(1,2) height_balL[OF _ _ True] by simp
        thus ?thesis
        proof
          assume ?A with 2 ‹x < a› show ?thesis by (auto)
        next
```

```
       assume ?B with 2 Node(2) True ‹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 2 Node(3,4) ‹¬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 2 Node(3) height_balR[OF _ _ True] by simp
      thus ?thesis
      proof
        assume ?A with 2 ‹¬x < a› show ?thesis by (auto)
      next
       assume ?B with 2 Node(4) True ‹¬x < a› show ?thesis by (simp)
arith
      qed
    qed
  qed
 qed
qed simp_all
```

Now an automatic proof without lemmas:

```
theorem avl_insert_auto: avl t ⟹
  avl(insert x t) ∧ height (insert x t) ∈ {height t, height t + 1}
apply (induction t rule: tree2_induct)

 apply (auto simp: balL_def balR_def node_def max_absorb2 split!: if_split
tree.split)
done
```

14.3.2 Deletion maintains AVL balance

```
lemma avl_split_max:
  ⟦ avl t; t ≠ Leaf ⟧ ⟹
  avl (fst (split_max t)) ∧
  height t ∈ {height(fst (split_max t)), height(fst (split_max t)) + 1}
by(induct t rule: split_max_induct)
   (auto simp: balL_def node_def max_absorb2 split!: prod.split if_split
tree.split)
```

Deletion maintains the AVL property:

```
theorem avl_delete:
  avl t ⟹ avl(delete x t)
  avl t ⟹ height t ∈ {height (delete x t), height (delete x t) + 1}
proof (induct t rule: tree2_induct)
  case (Node l a n r)
  case 1
  show ?case
  proof(cases x = a)
    case True thus ?thesis
      using 1 avl_split_max[of l] by (auto intro!: avl_balR split: prod.split)
  next
    case False thus ?thesis
      using Node 1 by (auto intro!: avl_balL avl_balR)
  qed
  case 2
  show ?case
  proof(cases x = a)
    case True thus ?thesis using 2 avl_split_max[of l]
    by(auto simp: balR_def max_absorb2 split!: if_splits prod.split tree.split)
  next
    case False
    show ?thesis
    proof(cases x<a)
      case True
      show ?thesis
      proof(cases height r = height (delete x l) + 2)
        case False
        thus ?thesis using 2 Node(1,2) ‹x < a› by(auto simp: balR_def)
      next
        case True
        thus ?thesis using height_balR[OF _ _ True, of a] 2 Node(1,2) ‹x
< a› by simp linarith
      qed
    next
      case False
      show ?thesis
      proof(cases height l = height (delete x r) + 2)
        case False
         thus ?thesis using 2 Node(3,4) ‹¬x < a› ‹x ≠ a› by(auto simp:
balL_def)
      next
```

```
        case True
        thus ?thesis
          using height_balL[OF _ _ True, of a] 2 Node(3,4) ‹¬x < a› ‹x ≠
a› by simp linarith
      qed
    qed
  qed
qed simp_all
```

A more automatic proof. Complete automation as for insertion seems hard due to resource requirements.

```
theorem avl_delete_auto:
  avl t ⟹ avl(delete x t)
  avl t ⟹ height t ∈ {height (delete x t), height (delete x t) + 1}
proof (induct t rule: tree2_induct)
  case (Node l a n r)
  case 1
  thus ?case
    using Node avl_split_max[of l] by (auto intro!: avl_balL avl_balR split:
prod.split)
  case 2
  show ?case
    using 2 Node avl_split_max[of l]
      by auto
          (auto simp: balL_def balR_def max_absorb1 max_absorb2  split!:
tree.splits prod.splits if_splits)
qed simp_all
```

14.4 Overall correctness

```
interpretation S: Set_by_Ordered
where empty = empty and isin = isin and insert = insert and delete =
delete
and inorder = inorder and inv = 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
```

14.5 Height-Size Relation

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

```
theorem avl_fib_bound:
  avl t ⟹ fib(height t + 2) ≤ size1 t
proof (induction rule: tree2_induct)
  case (Node l a h r)
  have 1: height l + 1 ≤ height r + 2 and 2: height r + 1 ≤ height l + 2
    using Node.prems by auto
  have fib (max (height l) (height r) + 3) ≤ size1 l + size1 r
  proof cases
    assume height l ≥ height r
    hence fib (max (height l) (height r) + 3) = fib (height l + 3)
      by(simp add: max_absorb1)
    also have ... = fib (height l + 2) + fib (height l + 1)
      by (simp add: numeral_eq_Suc)
    also have ... ≤ size1 l + fib (height l + 1)
      using Node by (simp)
    also have ... ≤ size1 r + size1 l
      using Node fib_mono[OF 1] by auto
    also have ... = size1 (Node l (a,h) r)
      by simp
    finally show ?thesis
      by (simp)
  next
    assume ¬ height l ≥ height r
    hence fib (max (height l) (height r) + 3) = fib (height r + 3)
      by(simp add: max_absorb1)
    also have ... = fib (height r + 2) + fib (height r + 1)
      by (simp add: numeral_eq_Suc)
    also have ... ≤ size1 r + fib (height r + 1)
      using Node by (simp)
    also have ... ≤ size1 r + size1 l
      using Node fib_mono[OF 2] by auto
    also have ... = size1 (Node l (a,h) r)
      by simp
    finally show ?thesis
      by (simp)
  qed
  also have ... = size1 (Node l (a,h) r)
    by simp
  finally show ?case by (simp del: fib.simps add: numeral_eq_Suc)
qed auto

lemma avl_fib_bound_auto: avl t ⟹ fib (height t + 2) ≤ size1 t
proof (induction t rule: tree2_induct)
  case Leaf thus ?case by (simp)
next
  case (Node l a h r)
  have 1: height l + 1 ≤ height r + 2 and 2: height r + 1 ≤ height l + 2
    using Node.prems by auto
  have left: height l ≥ height r ⟹ ?case (is ?asm ⟹ _)
    using Node fib_mono[OF 1] by (simp add: max.absorb1)
  have right: height l ≤ height r ⟹ ?case
    using Node fib_mono[OF 2] by (simp add: max.absorb2)
  show ?case using left right using Node.prems by simp linarith
qed
```

An exponential lower bound for *fib*:

```
lemma fib_lowerbound:
  defines φ ≡ (1 + sqrt 5) / 2
  shows real (fib(n+2)) ≥ φ ^ n
proof (induction n rule: fib.induct)
  case 1
  then show ?case by simp
next
  case 2
  then show ?case by (simp add: φ_def real_le_lsqrt)
next
  case (3 n)
  have φ ^ Suc (Suc n) = φ ^ 2 * φ ^ n
    by (simp add: field_simps power2_eq_square)
  also have ... = (φ + 1) * φ ^ n
    by (simp_all add: φ_def power2_eq_square field_simps)
  also have ... = φ ^ Suc n + φ ^ n
     by (simp add: field_simps)
  also have ... ≤ real (fib (Suc n + 2)) + real (fib (n + 2))
     by (intro add_mono 3.IH)
  finally show ?case by simp
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 ^ height t ≤ fib (height t + 2)
    unfolding φ_def by(rule fib_lowerbound)
  also have ... ≤ size1 t
    using avl_fib_bound[of 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 add: le_log_of_power)
  also have ... = (1/log 2 φ) ∗ log 2 (size1 t)
    by(simp add: log_base_change[of 2 φ])
  finally show ?thesis .
qed

end
```

15 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*) $\Longrightarrow$ *lookup t x* = *map_of* (*inorder t*) *x*
by(*induction t rule*: *tree2_induct*) (*auto simp*: *map_of_simps split*: *option.split*)

end

16 AVL Tree Implementation of Maps

theory *AVL_Map*
imports
  AVL_Set
  Lookup2
begin

```
fun update :: 'a::linorder ⇒ 'b ⇒ ('a*'b) tree_ht ⇒ ('a*'b) tree_ht 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) tree_ht ⇒ ('a*'b) tree_ht where
delete _ Leaf = Leaf |
delete x (Node l ((a,b), h) r) = (case cmp x a of
   EQ ⇒ if l = Leaf then r
        else let (l', ab') = split_max l in balR l' ab' r |
   LT ⇒ balR (delete x l) (a,b) r |
   GT ⇒ balL l (a,b) (delete x r))
```

16.1 Functional Correctness

theorem *inorder_update*:
  sorted1(*inorder t*) $\Longrightarrow$ *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*) $\Longrightarrow$ *inorder* (*delete x t*) = *del_list x* (*inorder t*)
by(*induction t*)
  (*auto simp*: *del_list_simps inorder_balL inorder_balR*
     inorder_split_maxD split: *prod.splits*)

16.2 AVL invariants

16.2.1 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 intro!: avl_balL)
    next
      case False with eq2 1 ‹x≠a› show ?thesis by (auto intro!: 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› height_balL[OF _ _ True] by simp
        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› height_balR[OF _ _ True] by simp
        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
```

16.2.2 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 rule: tree2_induct)
  case (Node l ab h r)
  obtain a b where [simp]: ab = (a,b) by fastforce
  case 1
  show ?case
  proof(cases x = a)
    case True with Node 1 show ?thesis
    using avl_split_max[of l] by (auto intro!: avl_balR split: prod.split)
  next
    case False
    show ?thesis
    proof(cases x<a)
      case True with Node 1 show ?thesis by (auto intro!: avl_balR)
    next
     case False with Node 1 ‹x≠a› show ?thesis by (auto intro!: avl_balL)
    qed
  qed
  case 2
  show ?case
  proof(cases x = a)
    case True then show ?thesis using 1 avl_split_max[of l]
     by(auto simp: balR_def max_absorb2 split!: if_splits prod.split tree.split)
  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
       thus ?thesis using height_balR[OF _ _ True, of ab] 2 Node(1,2) ‹x
< a› by simp linarith
      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
        thus ?thesis
          using height_balL[OF _ _ True, of ab] 2 Node(3,4) ‹¬x < a› ‹x
≠ a› by auto
      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

17 AVL Tree with Balance Factors (1)

theory *AVL_Bal_Set*
imports
  Cmp
  Isin2
begin

This version detects height increase/decrease from above via the change in balance factors.

datatype *bal* = *Lh* | *Bal* | *Rh*

type_synonym *'a tree_bal* = (*'a* ∗ *bal*) *tree*

Invariant:

fun *avl* :: *'a tree_bal* ⇒ *bool* **where**
avl Leaf = *True* |
avl (*Node l* (*a*,*b*) *r*) =
  ((*case b of*
    Bal ⇒ *height r* = *height l* |

```
    Lh ⇒ height l = height r + 1 |
    Rh ⇒ height r = height l + 1)
  ∧ avl l ∧ avl r)
```

17.1 Code

```
fun is_bal where
is_bal (Node l (a,b) r) = (b = Bal)

fun incr where
incr t t' = (t = Leaf ∨ is_bal t ∧ ¬ is_bal t')

fun rot2 where
rot2 A a B c C = (case B of
  (Node B1 (b, bb) B2) ⇒
    let b1 = if bb = Rh then Lh else Bal;
        b2 = if bb = Lh then Rh else Bal
    in Node (Node A (a,b1) B1) (b,Bal) (Node B2 (c,b2) C))

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

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

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

r′)

fun *decr* **where**
decr t t′ = (*t* ≠ *Leaf* ∧ (*t′* = *Leaf* ∨ ¬ *is_bal t* ∧ *is_bal t′*))

fun *split_max* :: *′a tree_bal* ⇒ *′a tree_bal* ∗ *′a* **where**
```
split_max (Node l (a, ba) r) =
  (if r = Leaf then (l,a)
   else let (r′,a′) = split_max r;
          t′ = if decr r r′ then balL l a ba r′ else Node l (a,ba) r′
       in (t′, a′))
```

fun *delete* :: *′a::linorder* ⇒ *′a tree_bal* ⇒ *′a tree_bal* **where**
```
delete _ Leaf = Leaf |
delete x (Node l (a, ba) r) =
  (case cmp x a of
     EQ ⇒ if l = Leaf then r
          else let (l′, a′) = split_max l in
              if decr l l′ then balR l′ a′ ba r else Node l′ (a′,ba) r |
     LT ⇒ let l′ = delete x l in if decr l l′ then balR l′ a ba r else Node l′
(a,ba) r |
     GT ⇒ let r′ = delete x r in if decr r r′ then balL l a ba r′ else Node l
(a,ba) r′)
```

17.2 Proofs

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

lemmas *splits* = *if_splits tree.splits bal.splits*

declare *Let_def* [*simp*]

17.2.1 Proofs about insertion

lemma *avl_insert*: *avl t* ⟹
  avl(*insert x t*) ∧
  height(*insert x t*) = *height t* + (*if incr t* (*insert x t*) *then 1 else 0*)
apply(*induction x t rule*: *insert.induct*)
apply(*auto split*!: *splits*)
done

The following two auxiliary lemma merely simplify the proof of *inorder_insert*.

lemma [*simp*]: [] ≠ *ins_list x xs*
by(*cases xs*) *auto*

lemma [*simp*]: *avl t* $\Longrightarrow$ *insert x t* $\neq$ ⟨*l*, (*a*, *Rh*), ⟨⟩⟩ $\wedge$ *insert x t* $\neq$ ⟨⟨⟩, (*a*, *Lh*), *r*⟩
by(*drule avl_insert*[*of _ x*]) (*auto split*: *splits*)

theorem *inorder_insert*:
  ⟦ *avl t*; *sorted*(*inorder t*) ⟧ $\Longrightarrow$ *inorder*(*insert x t*) = *ins_list x* (*inorder t*)
apply(*induction t*)
apply (*auto simp*: *ins_list_simps split*!: *splits*)
done

17.2.2 Proofs about deletion

lemma *inorder_balR*:
  ⟦ *ba* = *Rh* $\longrightarrow$ *r* $\neq$ *Leaf*; *avl r* ⟧
  $\Longrightarrow$ *inorder* (*balR l a ba r*) = *inorder l* @ *a* # *inorder r*
by (*auto split*: *splits*)

lemma *inorder_balL*:
  ⟦ *ba* = *Lh* $\longrightarrow$ *l* $\neq$ *Leaf*; *avl l* ⟧
  $\Longrightarrow$ *inorder* (*balL l a ba r*) = *inorder l* @ *a* # *inorder r*
by (*auto split*: *splits*)

lemma *height_1_iff*: *avl t* $\Longrightarrow$ *height t* = *Suc 0* $\longleftrightarrow$ ($\exists$ *x*. *t* = *Node Leaf* (*x*,*Bal*) *Leaf*)
by(*cases t*) (*auto split*: *splits prod.splits*)

lemma *avl_split_max*:
  ⟦ *split_max t* = (*t'*,*a*); *avl t*; *t* $\neq$ *Leaf* ⟧ $\Longrightarrow$
   avl t' $\wedge$ *height t* = *height t'* + (*if decr t t' then 1 else 0*)
apply(*induction t arbitrary*: *t' a rule*: *split_max_induct*)
 apply(*auto simp*: *max_absorb1 max_absorb2 height_1_iff split*!: *splits prod.splits*)
done

lemma *avl_delete*: *avl t* $\Longrightarrow$
  avl (*delete x t*) $\wedge$
  height t = *height* (*delete x t*) + (*if decr t* (*delete x t*) *then 1 else 0*)
apply(*induction x t rule*: *delete.induct*)
 apply(*auto simp*: *max_absorb1 max_absorb2 height_1_iff dest*: *avl_split_max split*!: *splits prod.splits*)
done

lemma *inorder_split_maxD*:
  ⟦ *split_max t* = (*t*′,*a*); *t* ≠ *Leaf*; *avl t* ⟧ ⟹
   inorder t′ @ [*a*] = *inorder t*
apply(*induction t arbitrary*: *t*′ *rule*: *split_max.induct*)
 apply(*fastforce split*!: *splits prod.splits*)
apply *simp*
done

lemma *neq_Leaf_if_height_neq_0*: *height t* ≠ *0* ⟹ *t* ≠ *Leaf*
by *auto*

lemma *split_max_Leaf*: ⟦ *t* ≠ *Leaf*; *avl t* ⟧ ⟹ *split_max t* = (⟨⟩, *x*) ⟷ *t* = *Node Leaf* (*x*,*Bal*) *Leaf*
by(*cases t*) (*auto split*: *splits prod.splits*)

theorem *inorder_delete*:
  ⟦ *avl t*; *sorted*(*inorder t*) ⟧ ⟹ *inorder* (*delete x t*) = *del_list x* (*inorder t*)
apply(*induction t rule*: *tree2_induct*)
apply(*auto simp*: *del_list_simps inorder_balR inorder_balL avl_delete inorder_split_maxD*
                  split_max_Leaf neq_Leaf_if_height_neq_0
          simp del: *balL.simps balR.simps split*!: *splits prod.splits*)
done

17.2.3 Set Implementation

interpretation *S*: *Set_by_Ordered*
where *empty* = *Leaf* **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*)
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*)
next
  case *6* **thus** *?case* **by** (*simp add*: *avl_insert*)

```
next
  case 7 thus ?case by (simp add: avl_delete)
qed

end
```

18 AVL Tree with Balance Factors (2)

```
theory AVL_Bal2_Set
imports
  Cmp
  Isin2
begin
```

This version passes a flag (*Same*/*Diff*) back up to signal if the height changed.

```
datatype bal = Lh | Bal | Rh

type_synonym 'a tree_bal = ('a * bal) tree
```

Invariant:

```
fun avl :: 'a tree_bal ⇒ bool where
avl Leaf = True |
avl (Node l (a,b) r) =
  ((case b of
    Bal ⇒ height r = height l |
    Lh ⇒ height l = height r + 1 |
    Rh ⇒ height r = height l + 1)
  ∧ avl l ∧ avl r)
```

18.1 Code

```
datatype 'a alt = Same 'a | Diff 'a

type_synonym 'a tree_bal2 = 'a tree_bal alt

fun tree :: 'a alt ⇒ 'a where
tree(Same t) = t |
tree(Diff t) = t

fun rot2 where
rot2 A a B c C = (case B of
  (Node B_1 (b, bb) B_2) ⇒
    let b_1 = if bb = Rh then Lh else Bal;
    $b_2$ = if bb = Lh then Rh else Bal
  in Node (Node A (a,$b_1$) $B_1$) (b,Bal) (Node $B_2$ (c,$b_2$) C))

fun balL :: 'a tree_bal2 ⇒ 'a ⇒ bal ⇒ 'a tree_bal ⇒ 'a tree_bal2 where
balL AB' c bc C = (case AB' of
   Same AB ⇒ Same (Node AB (c,bc) C) |
   Diff AB ⇒ (case bc of
     Bal ⇒ Diff (Node AB (c,Lh) C) |
     Rh ⇒ Same (Node AB (c,Bal) C) |
     Lh ⇒ (case AB of
       Node A (a,Lh) B ⇒ Same(Node A (a,Bal) (Node B (c,Bal) C)) |
       Node A (a,Rh) B ⇒ Same(rot2 A a B c C))))

fun balR :: 'a tree_bal ⇒ 'a ⇒ bal ⇒ 'a tree_bal2 ⇒ 'a tree_bal2 where
balR A a ba BC' = (case BC' of
   Same BC ⇒ Same (Node A (a,ba) BC) |
   Diff BC ⇒ (case ba of
     Bal ⇒ Diff (Node A (a,Rh) BC) |
     Lh ⇒ Same (Node A (a,Bal) BC) |
     Rh ⇒ (case BC of
       Node B (c,Rh) C ⇒ Same(Node (Node A (a,Bal) B) (c,Bal) C) |
       Node B (c,Lh) C ⇒ Same(rot2 A a B c C))))

fun ins :: 'a::linorder ⇒ 'a tree_bal ⇒ 'a tree_bal2 where
ins x Leaf = Diff(Node Leaf (x, Bal) Leaf) |
ins x (Node l (a, b) r) = (case cmp x a of
   EQ ⇒ Same(Node l (a, b) r) |
   LT ⇒ balL (ins x l) a b r |
   GT ⇒ balR l a b (ins x r))

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

fun baldR :: 'a tree_bal ⇒ 'a ⇒ bal ⇒ 'a tree_bal2 ⇒ 'a tree_bal2 where
baldR AB c bc C' = (case C' of
   Same C ⇒ Same (Node AB (c,bc) C) |
   Diff C ⇒ (case bc of
     Bal ⇒ Same (Node AB (c,Lh) C) |
     Rh ⇒ Diff (Node AB (c,Bal) C) |
     Lh ⇒ (case AB of
       Node A (a,Lh) B ⇒ Diff(Node A (a,Bal) (Node B (c,Bal) C)) |
       Node A (a,Bal) B ⇒ Same(Node A (a,Rh) (Node B (c,Lh) C)) |
       Node A (a,Rh) B ⇒ Diff(rot2 A a B c C))))
```

```
fun baldL :: 'a tree_bal2 ⇒ 'a ⇒ bal ⇒ 'a tree_bal ⇒ 'a tree_bal2 where
baldL A' a ba BC = (case A' of
   Same A ⇒ Same (Node A (a,ba) BC) |
   Diff A ⇒ (case ba of
     Bal ⇒ Same (Node A (a,Rh) BC) |
     Lh ⇒ Diff (Node A (a,Bal) BC) |
     Rh ⇒ (case BC of
       Node B (c,Rh) C ⇒ Diff(Node (Node A (a,Bal) B) (c,Bal) C) |
       Node B (c,Bal) C ⇒ Same(Node (Node A (a,Rh) B) (c,Lh) C) |
       Node B (c,Lh) C ⇒ Diff(rot2 A a B c C))))

fun split_max :: 'a tree_bal ⇒ 'a tree_bal2 ∗ 'a where
split_max (Node l (a, ba) r) =
  (if r = Leaf then (Diff l,a) else let (r',a') = split_max r in (baldR l a ba
r', a'))

fun del :: 'a::linorder ⇒ 'a tree_bal ⇒ 'a tree_bal2 where
del _ Leaf = Same Leaf |
del x (Node l (a, ba) r) =
  (case cmp x a of
     EQ ⇒ if l = Leaf then Diff r
           else let (l', a') = split_max l in baldL l' a' ba r |
     LT ⇒ baldL (del x l) a ba r |
     GT ⇒ baldR l a ba (del x r))

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

lemmas split_max_induct = split_max.induct[case_names Node Leaf]

lemmas splits = if_splits tree.splits alt.splits bal.splits
```

18.2 Proofs

18.2.1 Proofs about insertion

```
lemma avl_ins_case: avl t ⟹ case ins x t of
   Same t' ⇒ avl t' ∧ height t' = height t |
   Diff t' ⇒ avl t' ∧ height t' = height t + 1 ∧
      (∀ l a r. t' = Node l (a,Bal) r ⟶ a = x ∧ l = Leaf ∧ r = Leaf)
apply(induction x t rule: ins.induct)
apply(auto simp: max_absorb1 split!: splits)
done
```

corollary *avl_insert*: *avl t* $\Longrightarrow$ *avl*(*insert x t*)
using *avl_ins_case*[*of t x*] **by** (*simp add*: *insert_def split*: *splits*)

lemma *ins_Diff*[*simp*]: *avl t* $\Longrightarrow$
  ins x t $\neq$ *Diff Leaf* $\wedge$
  (*ins x t* = *Diff* (*Node l* (*a*,*Bal*) *r*) $\longleftrightarrow$ *t* = *Leaf* $\wedge$ *a* = *x* $\wedge$ *l*=*Leaf* $\wedge$ *r*=*Leaf*) $\wedge$
  ins x t $\neq$ *Diff* (*Node l* (*a*,*Rh*) *Leaf*) $\wedge$
  ins x t $\neq$ *Diff* (*Node Leaf* (*a*,*Lh*) *r*)
by(*drule avl_ins_case*[*of _ x*]) (*auto split*: *splits*)

theorem *inorder_ins*:
  $[\![$ *avl t*; *sorted*(*inorder t*) $]\!]$ $\Longrightarrow$ *inorder*(*tree*(*ins x t*)) = *ins_list x* (*inorder t*)
apply(*induction t*)
apply (*auto simp*: *ins_list_simps split*!: *splits*)
done

18.2.2 Proofs about deletion

lemma *inorder_baldL*:
  $[\![$ *ba* = *Rh* $\longrightarrow$ *r* $\neq$ *Leaf*; *avl r* $]\!]$
  $\Longrightarrow$ *inorder* (*tree*(*baldL l a ba r*)) = *inorder* (*tree l*) @ *a* # *inorder r*
by (*auto split*: *splits*)

lemma *inorder_baldR*:
  $[\![$ *ba* = *Lh* $\longrightarrow$ *l* $\neq$ *Leaf*; *avl l* $]\!]$
   $\Longrightarrow$ *inorder* (*tree*(*baldR l a ba r*)) = *inorder l* @ *a* # *inorder* (*tree r*)
by (*auto split*: *splits*)

lemma *avl_split_max*:
  $[\![$ *split_max t* = (*t'*,*a*); *avl t*; *t* $\neq$ *Leaf* $]\!]$ $\Longrightarrow$ *case t' of*
   Same t' $\Rightarrow$ *avl t'* $\wedge$ *height t* = *height t'* |
   Diff t' $\Rightarrow$ *avl t'* $\wedge$ *height t* = *height t'* + *1*
apply(*induction t arbitrary*: *t' a rule*: *split_max_induct*)
 apply(*fastforce simp*: *max_absorb1 max_absorb2 split*!: *splits prod.splits*)
apply *simp*
done

lemma *avl_del_case*: *avl t* $\Longrightarrow$ *case del x t of*
   Same t' $\Rightarrow$ *avl t'* $\wedge$ *height t* = *height t'* |
   Diff t' $\Rightarrow$ *avl t'* $\wedge$ *height t* = *height t'* + *1*

apply(*induction x t rule*: *del.induct*)
 apply(*auto simp*: *max_absorb1 max_absorb2 dest*: *avl_split_max split*!: *splits prod.splits*)
done

corollary *avl_delete*: *avl t* $\Longrightarrow$ *avl*(*delete x t*)
using *avl_del_case*[*of t x*] **by**(*simp add*: *delete_def split*: *splits*)

lemma *inorder_split_maxD*:
  ⟦ *split_max t* = (*t'*,*a*); *t* ≠ *Leaf*; *avl t* ⟧ $\Longrightarrow$
   inorder (*tree t'*) @ [*a*] = *inorder t*
apply(*induction t arbitrary*: *t' rule*: *split_max.induct*)
 apply(*fastforce split*!: *splits prod.splits*)
apply *simp*
done

lemma *neq_Leaf_if_height_neq_0*[*simp*]: *height t* ≠ *0* $\Longrightarrow$ *t* ≠ *Leaf*
by *auto*

theorem *inorder_del*:
  ⟦ *avl t*; *sorted*(*inorder t*) ⟧ $\Longrightarrow$ *inorder* (*tree*(*del x t*)) = *del_list x* (*inorder t*)
apply(*induction t rule*: *tree2_induct*)
apply(*auto simp*: *del_list_simps inorder_baldL inorder_baldR avl_delete inorder_split_maxD*
          simp del: *baldR.simps baldL.simps split*!: *splits prod.splits*)
done

18.2.3 Set Implementation

interpretation *S*: *Set_by_Ordered*
where *empty* = *Leaf* **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*)
next
  case *2* **thus** *?case* **by**(*simp add*: *isin_set_inorder*)
next
  case *3* **thus** *?case* **by**(*simp add*: *inorder_ins insert_def*)
next
  case *4* **thus** *?case* **by**(*simp add*: *inorder_del delete_def*)
next

```
  case 5 thus ?case by (simp)
next
  case 6 thus ?case by (simp add: avl_insert)
next
  case 7 thus ?case by (simp add: avl_delete)
qed

end
```

19 Height-Balanced Trees

```
theory Height_Balanced_Tree
imports
  Cmp
  Isin2
begin
```

Height-balanced trees (HBTs) can be seen as a generalization of AVL trees. The code and the proofs were obtained by small modifications of the AVL theories. This is an implementation of sets via HBTs.

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

definition empty :: 'a tree_ht where
empty = Leaf
```

The maximal amount by which the height of two siblings may differ:

```
locale HBT =
fixes m :: nat
assumes [arith]: m > 0
begin
```

Invariant:

```
fun hbt :: 'a tree_ht ⇒ bool where
hbt Leaf = True |
hbt (Node l (a,n) r) =
 (abs(int(height l) − int(height r)) ≤ int(m) ∧
  n = max (height l) (height r) + 1 ∧ hbt l ∧ hbt r)

fun ht :: 'a tree_ht ⇒ nat where
ht Leaf = 0 |
ht (Node l (a,n) r) = n

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

```
definition balL :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht where
balL AB b C =
  (if ht AB = ht C + m + 1 then
     case AB of
       Node A (a, _) B ⇒
         if ht A ≥ ht B then node A a (node B b C)
         else
           case B of
             Node B1 (ab, _) B2 ⇒ node (node A a B1) ab (node B2 b C)
   else node AB b C)

definition balR :: 'a tree_ht ⇒ 'a ⇒ 'a tree_ht ⇒ 'a tree_ht where
balR A a BC =
   (if ht BC = ht A + m + 1 then
      case BC of
        Node B (b, _) C ⇒
          if ht B ≤ ht C then node (node A a B) b C
          else
            case B of
              Node B1 (ab, _) B2 ⇒ node (node A a B1) ab (node B2 b C)
  else node A a BC)

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

fun split_max :: 'a tree_ht ⇒ 'a tree_ht * '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 delete :: 'a::linorder ⇒ 'a tree_ht ⇒ 'a tree_ht where
delete _ Leaf = Leaf |
delete x (Node l (a, n) r) =
  (case cmp x a of
     EQ ⇒ if l = Leaf then r
           else let (l', a') = split_max l in balR l' a' r |
     LT ⇒ balR (delete x l) a r |
     GT ⇒ balL l a (delete x r))
```

19.1 Functional Correctness Proofs

19.1.1 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)

19.1.2 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)

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_split_maxD
split: prod.splits)

19.2 Invariant preservation

19.2.1 Insertion maintains balance

declare Let_def [simp]

lemma ht_height[simp]: hbt t ⟹ ht t = height t
by (cases t rule: tree2_cases) simp_all

First, a fast but relatively manual proof with many lemmas:

lemma height_balL:
  ⟦ hbt l; hbt r; height l = height r + m + 1 ⟧ ⟹
   height (balL l a r) ∈ {height r + m + 1, height r + m + 2}
by (auto simp:node_def balL_def split:tree.split)

```
lemma height_balR:
  ⟦ hbt l; hbt r; height r = height l + m + 1 ⟧ ⟹
   height (balR l a r) ∈ {height l + m + 1, height l + m + 2}
by(auto simp add:node_def balR_def split:tree.split)

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

lemma height_balL2:
  ⟦ hbt l; hbt r; height l ≠ height r + m + 1 ⟧ ⟹
   height (balL l a r) = 1 + max (height l) (height r)
by (simp_all add: balL_def)

lemma height_balR2:
  ⟦ hbt l;  hbt r;  height r ≠ height l + m + 1 ⟧ ⟹
   height (balR l a r) = 1 + max (height l) (height r)
by (simp_all add: balR_def)

lemma hbt_balL:
  ⟦ hbt l; hbt r; height r − m ≤ height l ∧ height l ≤ height r + m + 1 ⟧
⟹ hbt(balL l a r)
by(auto simp: balL_def node_def max_def split!: if_splits tree.split)

lemma hbt_balR:
  ⟦ hbt l; hbt r; height l − m ≤ height r ∧ height r ≤ height l + m + 1 ⟧
⟹ hbt(balR l a r)
by(auto simp: balR_def node_def max_def split!: if_splits tree.split)
```

Insertion maintains *hbt*. Requires simultaneous proof.

```
theorem hbt_insert:
  hbt t ⟹ hbt(insert x t)
  hbt t ⟹ height (insert x t) ∈ {height t, height t + 1}
proof (induction t rule: tree2_induct)
  case (Node l a _ 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 1 Node(1,2) show ?thesis by (auto intro!: hbt_balL)
    next
      case False with 1 Node(3,4) ‹x≠a› show ?thesis by (auto intro!:
hbt_balR)
    qed
  qed
  case 2
  show ?case
  proof(cases x = a)
    case True with 2 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 + m + 1)
        case False with 2 Node(1,2) ‹x < a› show ?thesis by (auto simp:
height_balL2)
      next
        case True
        hence (height (balL (insert x l) a r) = height r + m + 1) ∨
          (height (balL (insert x l) a r) = height r + m + 2) (is ?A ∨ ?B)
          using 2 Node(1,2) height_balL[OF _ _ True] by simp
        thus ?thesis
        proof
          assume ?A with 2 Node(2) True ‹x < a› show ?thesis by (auto)
        next
          assume ?B with 2 Node(2) True ‹x < a› show ?thesis by (simp)
arith
        qed
      qed
    next
      case False
      show ?thesis
      proof(cases height (insert x r) = height l + m + 1)
        case False with 2 Node(3,4) ‹¬x < a› show ?thesis by (auto simp:
height_balR2)
      next
        case True
        hence (height (balR l a (insert x r)) = height l + m + 1) ∨
          (height (balR l a (insert x r)) = height l + m + 2) (is ?A ∨ ?B)
          using Node 2 height_balR[OF _ _ True] by simp
        thus ?thesis
        proof
```

assume *?A* **with** *2 Node(4) True* ‹¬*x* < *a*› **show** *?thesis* **by** (*auto*)
next
assume *?B* **with** *2 Node(4) True* ‹¬*x* < *a*› **show** *?thesis* **by** (*simp*) *arith*
qed
qed
qed
qed
qed *simp_all*

Now an automatic proof without lemmas:

theorem *hbt_insert_auto*: *hbt t* $\Longrightarrow$
hbt(*insert x t*) $\wedge$ *height* (*insert x t*) $\in$ {*height t*, *height t* + *1*}
apply (*induction t rule*: *tree2_induct*)

apply (*auto simp*: *balL_def balR_def node_def max_absorb1 max_absorb2 split*!: *if_split tree.split*)
done

19.2.2 Deletion maintains balance

lemma *hbt_split_max*:
⟦ *hbt t*; *t* ≠ *Leaf* ⟧ $\Longrightarrow$
hbt (*fst* (*split_max t*)) $\wedge$
height t $\in$ {*height*(*fst* (*split_max t*)), *height*(*fst* (*split_max t*)) + *1*}
by(*induct t rule*: *split_max_induct*)
(*auto simp*: *balL_def node_def max_absorb2 split*!: *prod.split if_split tree.split*)

Deletion maintains *hbt*:

theorem *hbt_delete*:
hbt t $\Longrightarrow$ *hbt*(*delete x t*)
hbt t $\Longrightarrow$ *height t* $\in$ {*height* (*delete x t*), *height* (*delete x t*) + *1*}
proof (*induct t rule*: *tree2_induct*)
case (*Node l a n r*)
case *1*
thus *?case*
using *Node hbt_split_max*[*of l*] **by** (*auto intro*!: *hbt_balL hbt_balR split*: *prod.split*)
case *2*
show *?case*
proof(*cases x* = *a*)
case *True* **then show** *?thesis* **using** *1 hbt_split_max*[*of l*]
by(*auto simp*: *balR_def max_absorb2 split*!: *if_splits prod.split tree.split*)

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

simp: *balL_def split*: *if_splits*)
        qed
      qed
    qed
  qed
qed *simp_all*

A more automatic proof. Complete automation as for insertion seems hard due to resource requirements.

theorem *hbt_delete_auto*:
  hbt t $\Longrightarrow$ *hbt*(*delete x t*)
  hbt t $\Longrightarrow$ *height t* $\in$ {*height* (*delete x t*), *height* (*delete x t*) + *1*}
proof (*induct t rule*: *tree2_induct*)
  case (*Node l a n r*)
  case *1*
  thus *?case*
   using *Node hbt_split_max*[*of l*] **by** (*auto intro*!: *hbt_balL hbt_balR split*: *prod.split*)
  case *2*
  show *?case*
  proof(*cases x* = *a*)
    case *True* **thus** *?thesis*
      using *2 hbt_split_max*[*of l*]
     by(*auto simp*: *balR_def max_absorb2 split*!: *if_splits prod.split tree.split*)
  next
    case *False* **thus** *?thesis*
       using *height_balL*[*of l delete x r a*] *height_balR*[*of delete x l r a*] *2 Node*
        by(*auto simp*: *balL_def balR_def split*!: *if_split*)
  qed
qed *simp_all*

19.3 Overall correctness

interpretation *S*: *Set_by_Ordered*
where *empty* = *empty* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = *hbt*
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: hbt_insert(1))
next
  case 7 thus ?case by (simp add: hbt_delete(1))
qed

end

end
```

20 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 a t2) b t3) c t4 = R (B t1 a t2) b (B t3 c t4) |
baliL (R t1 a (R t2 b t3)) c t4 = R (B t1 a t2) b (B t3 c t4) |
baliL t1 a t2 = B t1 a t2

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

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 a t2) b t3 = R (B t1 a t2) b t3 |
```

baldL t1 a (*B t2 b t3*) = *baliR t1 a* (*R t2 b t3*) |
baldL t1 a (*R* (*B t2 b t3*) *c t4*) = *R* (*B t1 a t2*) *b* (*baliR t3 c* (*paint Red t4*)) |
baldL t1 a t2 = *R t1 a t2*

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

fun *join* :: *'a rbt* ⇒ *'a rbt* ⇒ *'a rbt* **where**

```
join Leaf t = t |
join t Leaf = t |
join (R t1 a t2) (R t3 c t4) =
  (case join t2 t3 of
     R u2 b u3 ⇒ (R (R t1 a u2) b (R u3 c t4)) |
     t23 ⇒ R t1 a (R t23 c t4)) |
join (B t1 a t2) (B t3 c t4) =
  (case join t2 t3 of
     R u2 b u3 ⇒ R (B t1 a u2) b (B u3 c t4) |
     t23 ⇒ baldL t1 a (B t23 c t4)) |
join t1 (R t2 a t3) = R (join t1 t2) a t3 |
join (R t1 a t2) t3 = R t1 a (join t2 t3)
```

end

21 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 ⇒ join l r)

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

21.1 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*:
  $sorted(inorder\ t) \Longrightarrow inorder(ins\ x\ t) = ins\_list\ x\ (inorder\ t)$
by(*induction x t rule*: *ins.induct*)
  (*auto simp*: *ins_list_simps inorder_baliL inorder_baliR*)

lemma *inorder_insert*:
  $sorted(inorder\ t) \Longrightarrow inorder(insert\ x\ t) = ins\_list\ x\ (inorder\ t)$
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_join*:
  $inorder(join\ l\ r) = inorder\ l\ @\ inorder\ r$
by(*induction l r rule*: *join.induct*)
  (*auto simp*: *inorder_baldL inorder_baldR split*: *tree.split color.split*)

lemma *inorder_del*:
 $sorted(inorder\ t) \Longrightarrow inorder(del\ x\ t) = del\_list\ x\ (inorder\ t)$
by(*induction x t rule*: *del.induct*)
  (*auto simp*: *del_list_simps inorder_join inorder_baldL inorder_baldR*)

lemma *inorder_delete*:
  $sorted(inorder\ t) \Longrightarrow inorder(delete\ x\ t) = del\_list\ x\ (inorder\ t)$
by (*auto simp*: *delete_def inorder_del inorder_paint*)

21.2 Structural invariants

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

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*) =
  ((*c* = *Red* ⟶ *color l* = *Black* ∧ *color r* = *Black*) ∧ *invc l* ∧ *invc r*)

Weaker version:

abbreviation *invc2* :: *'a rbt* ⇒ *bool* **where**
invc2 t ≡ *invc*(*paint Black t*)

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

lemma *invc2I*: *invc t* ⟹ *invc2 t*
by (*cases t rule*: *tree2_cases*) *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 *paint2*: *paint c2* (*paint c1 t*) = *paint c2 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*:

$\llbracket$ *invh l*; *invh r*; *bheight l* = *bheight r* $\rrbracket \Longrightarrow$ *invh* (*baliL l a r*)
by (*induct l a r rule*: *baliL.induct*) *auto*

lemma *invh_baliR*:
$\llbracket$ *invh l*; *invh r*; *bheight l* = *bheight r* $\rrbracket \Longrightarrow$ *invh* (*baliR l a r*)
by (*induct l a r rule*: *baliR.induct*) *auto*

All in one:

lemma *inv_baliR*: $\llbracket$ *invh l*; *invh r*; *invc l*; *invc2 r*; *bheight l* = *bheight r* $\rrbracket$ $\Longrightarrow$ *invc* (*baliR l a r*) $\wedge$ *invh* (*baliR l a r*) $\wedge$ *bheight* (*baliR l a r*) = *Suc* (*bheight l*)
by (*induct l a r rule*: *baliR.induct*) *auto*

lemma *inv_baliL*: $\llbracket$ *invh l*; *invh r*; *invc2 l*; *invc r*; *bheight l* = *bheight r* $\rrbracket$ $\Longrightarrow$ *invc* (*baliL l a r*) $\wedge$ *invh* (*baliL l a r*) $\wedge$ *bheight* (*baliL l a r*) = *Suc* (*bheight l*)
by (*induct l a r rule*: *baliL.induct*) *auto*

21.2.1 Insertion

lemma *invc_ins*: *invc t* $\longrightarrow$ *invc2* (*ins x t*) $\wedge$ (*color t* = *Black* $\longrightarrow$ *invc* (*ins x t*))
by (*induct x t rule*: *ins.induct*) (*auto simp*: *invc_baliL invc_baliR invc2I*)

lemma *invh_ins*: *invh t* $\Longrightarrow$ *invh* (*ins x t*) $\wedge$ *bheight* (*ins x t*) = *bheight t*
by(*induct x t rule*: *ins.induct*)
(*auto simp*: *invh_baliL invh_baliR bheight_baliL bheight_baliR*)

theorem *rbt_insert*: *rbt t* $\Longrightarrow$ *rbt* (*insert x t*)
by (*simp add*: *invc_ins invh_ins color_paint_Black invh_paint rbt_def insert_def*)

All in one:

lemma *inv_ins*: $\llbracket$ *invc t*; *invh t* $\rrbracket \Longrightarrow$
invc2 (*ins x t*) $\wedge$ (*color t* = *Black* $\longrightarrow$ *invc* (*ins x t*)) $\wedge$
invh(*ins x t*) $\wedge$ *bheight* (*ins x t*) = *bheight t*
by (*induct x t rule*: *ins.induct*) (*auto simp*: *inv_baliL inv_baliR invc2I*)

theorem *rbt_insert2*: *rbt t* $\Longrightarrow$ *rbt* (*insert x t*)
by (*simp add*: *inv_ins color_paint_Black invh_paint rbt_def insert_def*)

21.2.2 Deletion

lemma *bheight_paint_Red*:
color t = *Black* $\Longrightarrow$ *bheight* (*paint Red t*) = *bheight t* − *1*

by (*cases t*) *auto*

lemma *invh_baldL_invc*:
  $\llbracket$ *invh l*; *invh r*; *bheight l* + *1* = *bheight r*; *invc r* $\rrbracket$
  $\Longrightarrow$ *invh* (*baldL l a r*) $\wedge$ *bheight* (*baldL l a r*) = *bheight r*
by (*induct l a r rule*: *baldL.induct*)
  (*auto simp*: *invh_baliR invh_paint bheight_baliR bheight_paint_Red*)

lemma *invh_baldL_Black*:
  $\llbracket$ *invh l*; *invh r*; *bheight l* + *1* = *bheight r*; *color r* = *Black* $\rrbracket$
  $\Longrightarrow$ *invh* (*baldL l a r*) $\wedge$ *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*: $\llbracket$*invc2 l*; *invc r*; *color r* = *Black*$\rrbracket$ $\Longrightarrow$ *invc* (*baldL l a r*)
by (*induct l a r rule*: *baldL.induct*) (*simp_all add*: *invc_baliR*)

lemma *invc2_baldL*: $\llbracket$ *invc2 l*; *invc r* $\rrbracket$ $\Longrightarrow$ *invc2* (*baldL l a r*)
by (*induct l a r rule*: *baldL.induct*) (*auto simp*: *invc_baliR paint2 invc2I*)

lemma *invh_baldR_invc*:
  $\llbracket$ *invh l*; *invh r*; *bheight l* = *bheight r* + *1*; *invc l* $\rrbracket$
  $\Longrightarrow$ *invh* (*baldR l a r*) $\wedge$ *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*: $\llbracket$*invc l*; *invc2 r*; *color l* = *Black*$\rrbracket$ $\Longrightarrow$ *invc* (*baldR l a r*)
by (*induct l a r rule*: *baldR.induct*) (*simp_all add*: *invc_baliL*)

lemma *invc2_baldR*: $\llbracket$ *invc l*; *invc2 r* $\rrbracket$ $\Longrightarrow$*invc2* (*baldR l a r*)
by (*induct l a r rule*: *baldR.induct*) (*auto simp*: *invc_baliL paint2 invc2I*)

lemma *invh_join*:
  $\llbracket$ *invh l*; *invh r*; *bheight l* = *bheight r* $\rrbracket$
  $\Longrightarrow$ *invh* (*join l r*) $\wedge$ *bheight* (*join l r*) = *bheight l*
by (*induct l r rule*: *join.induct*)
  (*auto simp*: *invh_baldL_Black split*: *tree.splits color.splits*)

lemma *invc_join*:
  $\llbracket$ *invc l*; *invc r* $\rrbracket$ $\Longrightarrow$
  (*color l* = *Black* $\wedge$ *color r* = *Black* $\longrightarrow$ *invc* (*join l r*)) $\wedge$ *invc2* (*join l r*)
by (*induct l r rule*: *join.induct*)

(*auto simp*: *invc_baldL invc2I split*: *tree.splits color.splits*)

All in one:

lemma *inv_baldL*:
  $\llbracket$ *invh l*; *invh r*; *bheight l* + *1* = *bheight r*; *invc2 l*; *invc r* $\rrbracket$
  $\Longrightarrow$ *invh* (*baldL l a r*) $\wedge$ *bheight* (*baldL l a r*) = *bheight r*
  $\wedge$ *invc2* (*baldL l a r*) $\wedge$ (*color r* = *Black* $\longrightarrow$ *invc* (*baldL l a r*))
by (*induct l a r rule*: *baldL.induct*)
  (*auto simp*: *inv_baliR invh_paint bheight_baliR bheight_paint_Red paint2 invc2I*)

lemma *inv_baldR*:
  $\llbracket$ *invh l*; *invh r*; *bheight l* = *bheight r* + *1*; *invc l*; *invc2 r* $\rrbracket$
  $\Longrightarrow$ *invh* (*baldR l a r*) $\wedge$ *bheight* (*baldR l a r*) = *bheight l*
  $\wedge$ *invc2* (*baldR l a r*) $\wedge$ (*color l* = *Black* $\longrightarrow$ *invc* (*baldR l a r*))
by (*induct l a r rule*: *baldR.induct*)
  (*auto simp*: *inv_baliL invh_paint bheight_baliL bheight_paint_Red paint2 invc2I*)

lemma *inv_join*:
  $\llbracket$ *invh l*; *invh r*; *bheight l* = *bheight r*; *invc l*; *invc r* $\rrbracket$
  $\Longrightarrow$ *invh* (*join l r*) $\wedge$ *bheight* (*join l r*) = *bheight l*
  $\wedge$ *invc2* (*join l r*) $\wedge$ (*color l* = *Black* $\wedge$ *color r* = *Black* $\longrightarrow$ *invc* (*join l r*))
by (*induct l r rule*: *join.induct*)
  (*auto simp*: *invh_baldL_Black inv_baldL invc2I split*: *tree.splits color.splits*)

lemma *neq_LeafD*: *t* $\neq$ *Leaf* $\Longrightarrow$ $\exists$ *l x c r*. *t* = *Node l* (*x*,*c*) *r*
by(*cases t rule*: *tree2_cases*) *auto*

lemma *inv_del*: $\llbracket$ *invh t*; *invc t* $\rrbracket$ $\Longrightarrow$
   invh (*del x t*) $\wedge$
   (*color t* = *Red* $\longrightarrow$ *bheight* (*del x t*) = *bheight t* $\wedge$ *invc* (*del x t*)) $\wedge$
   (*color t* = *Black* $\longrightarrow$ *bheight* (*del x t*) = *bheight t* − *1* $\wedge$ *invc2* (*del x t*))
by(*induct x t rule*: *del.induct*)
  (*auto simp*: *inv_baldL inv_baldR inv_join dest*!: *neq_LeafD*)

theorem *rbt_delete*: *rbt t* $\Longrightarrow$ *rbt* (*delete x t*)
by (*metis delete_def rbt_def color_paint_Black inv_del 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
```

21.3 Height-Size Relation

```
lemma rbt_height_bheight_if: invc t ⟹ invh t ⟹
  height t ≤ 2 * bheight t + (if color t = Black then 0 else 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
```

22 Alternative Deletion in Red-Black Trees

theory *RBT_Set2*
imports *RBT_Set*
begin

This is a conceptually simpler version of deletion. Instead of the tricky *join* function this version follows the standard approach of replacing the deleted element (in function *del*) by the minimal element in its right subtree.

```
fun split_min :: 'a rbt ⇒ 'a × 'a rbt where
split_min (Node l (a, _) r) =
  (if l = Leaf then (a,r)
   else let (x,l') = split_min l
        in (x, if color l = Black then baldL l' a r else R l' a r))

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 ⇒ let l' = del x l in if l ≠ Leaf ∧ color l = Black
          then baldL l' a r else R l' a r |
     GT ⇒ let r' = del x r in if r ≠ Leaf ∧ color r = Black
          then baldR l a r' else R l a r' |
     EQ ⇒ if r = Leaf then l else let (a',r') = split_min r in
          if color r = Black then baldR l a' r' else R l a' r')
```

The first two *let*s speed up the automatic proof of *inv_del* below.

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

22.1 Functional Correctness Proofs

```
declare Let_def[simp]

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: inorder_baldL sorted_lems split: prod.splits if_splits)

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_baldL inorder_baldR split_minD split:
prod.splits)
```

lemma *inorder_delete*:
  $sorted(inorder\ t) \Longrightarrow inorder(delete\ x\ t) = del\_list\ x\ (inorder\ t)$
by (*auto simp*: *delete_def inorder_del inorder_paint*)

22.2 Structural invariants

lemma *neq_Red*[*simp*]: $(c \neq Red) = (c = Black)$
by (*cases c*) *auto*

22.2.1 Deletion

lemma *inv_split_min*: $\llbracket\ split\_min\ t = (x,t');\ t \neq Leaf;\ invh\ t;\ invc\ t\ \rrbracket$
$\Longrightarrow$
  $invh\ t' \wedge$
  $(color\ t = Red \longrightarrow bheight\ t' = bheight\ t \wedge invc\ t') \wedge$
  $(color\ t = Black \longrightarrow bheight\ t' = bheight\ t - 1 \wedge invc2\ t')$
apply(*induction t arbitrary*: *x t′ rule*: *split_min.induct*)
apply(*auto simp*: *inv_baldR inv_baldL invc2I dest*!: *neq_LeafD*
    split: *if_splits prod.splits*)
done

An automatic proof. It is quite brittle, e.g. inlining the *let*s in *RBT_Set2.del* breaks it.

lemma *inv_del*: $\llbracket\ invh\ t;\ invc\ t\ \rrbracket \Longrightarrow$
  $invh\ (del\ x\ t) \wedge$
  $(color\ t = Red \longrightarrow bheight\ (del\ x\ t) = bheight\ t \wedge invc\ (del\ x\ t)) \wedge$
  $(color\ t = Black \longrightarrow bheight\ (del\ x\ t) = bheight\ t - 1 \wedge invc2\ (del\ x\ t))$
apply(*induction x t rule*: *del.induct*)
apply(*auto simp*: *inv_baldR inv_baldL invc2I dest*!: *inv_split_min dest*: *neq_LeafD*
    split!: *prod.splits if_splits*)
done

A structured proof where one can see what is used in each case.

lemma *inv_del2*: $\llbracket\ invh\ t;\ invc\ t\ \rrbracket \Longrightarrow$
  $invh\ (del\ x\ t) \wedge$
  $(color\ t = Red \longrightarrow bheight\ (del\ x\ t) = bheight\ t \wedge invc\ (del\ x\ t)) \wedge$
  $(color\ t = Black \longrightarrow bheight\ (del\ x\ t) = bheight\ t - 1 \wedge invc2\ (del\ x\ t))$
proof(*induction x t rule*: *del.induct*)
  case (*1 x*)
  then show *?case* **by** *simp*
next
  case (*2 x l a c r*)
  note *if_split*[*split del*]
  show *?case*

```
  proof cases
    assume x < a
    show ?thesis
    proof cases
      assume l = Leaf thus ?thesis using ‹x < a› 2.prems by(auto)
    next
      assume l: l ≠ Leaf
      show ?thesis
      proof (cases color l)
        assume *: color l = Black
        hence bheight l > 0 using l neq_LeafD[of l] by auto
        thus ?thesis using ‹x < a› 2.IH(1) 2.prems inv_baldL[of del x l] *
l by(auto)
      next
        assume color l = Red
        thus ?thesis using ‹x < a› 2.prems 2.IH(1) by(auto)
      qed
    qed
  next
    assume ¬ x < a
    show ?thesis
    proof cases
      assume x > a
     show ?thesis using ‹a < x› 2.IH(2) 2.prems neq_LeafD[of r] inv_baldR[of
_ del x r]
          by(auto split: if_split)

    next
      assume ¬ x > a
      show ?thesis using 2.prems ‹¬ x < a› ‹¬ x > a›
        by(auto simp: inv_baldR invc2I dest!: inv_split_min dest: neq_LeafD
split: prod.split if_split)
    qed
  qed
qed

theorem rbt_delete: rbt t ⟹ rbt (delete x t)
by (metis delete_def rbt_def color_paint_Black inv_del 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


end
```

23 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 ⇒ join l r)

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

23.1 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_join 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*)

23.2 Structural invariants

23.2.1 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* $\Longrightarrow$ *rbt* (*update x y t*)
by (*simp add*: *invc_upd*(*2*) *invh_upd*(*1*) *color_paint_Black invh_paint rbt_def update_def*)

23.2.2 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_join invc_join)
  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 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

24 2-3 Trees

theory *Tree23*
imports *Main*
begin

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

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

fun *inorder* :: *'a tree23* ⇒ *'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

Completeness:

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

```
lemma ht_sz_if_complete: complete t ⟹ 2 ^ height t ≤ size t + 1
by (induction t) auto
```

end

25 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 upI = TI 'a tree23 | OF 'a tree23 'a 'a tree23

fun treeI :: 'a upI ⇒ 'a tree23 where
treeI (TI t) = t |
treeI (OF l a r) = Node2 l a r

fun ins :: 'a::linorder ⇒ 'a tree23 ⇒ 'a upI where
ins x Leaf = OF Leaf x Leaf |
ins x (Node2 l a r) =
  (case cmp x a of
    LT ⇒
      (case ins x l of
        TI l' => TI (Node2 l' a r) |
        OF l1 b l2 => TI (Node3 l1 b l2 a r)) |
    EQ ⇒ TI (Node2 l a r) |
    GT ⇒
      (case ins x r of
        TI r' => TI (Node2 l a r') |
        OF r1 b r2 => TI (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
        TI l' => TI (Node3 l' a m b r) |
        OF l1 c l2 => OF (Node2 l1 c l2) a (Node2 m b r)) |
    EQ ⇒ TI (Node3 l a m b r) |
    GT ⇒
      (case cmp x b of
        GT ⇒
          (case ins x r of
            TI r' => TI (Node3 l a m b r') |
            OF r1 c r2 => OF (Node2 l a m) b (Node2 r1 c r2)) |
        EQ ⇒ TI (Node3 l a m b r) |
        LT ⇒
          (case ins x m of
            TI m' => TI (Node3 l a m' b r) |
            OF m1 c m2 => OF (Node2 l a m1) c (Node2 m2 b r))))

hide_const insert

definition insert :: 'a::linorder ⇒ 'a tree23 ⇒ 'a tree23 where
insert x t = treeI(ins x t)

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

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

```
fun node21 :: 'a upD ⇒ 'a ⇒ 'a tree23 ⇒ 'a upD where
node21 (TD t1) a t2 = TD(Node2 t1 a t2) |
node21 (UF t1) a (Node2 t2 b t3) = UF(Node3 t1 a t2 b t3) |
node21 (UF t1) a (Node3 t2 b t3 c t4) = TD(Node2 (Node2 t1 a t2) b
(Node2 t3 c t4))
```

```
fun node22 :: 'a tree23 ⇒ 'a ⇒ 'a upD ⇒ 'a upD where
node22 t1 a (TD t2) = TD(Node2 t1 a t2) |
node22 (Node2 t1 b t2) a (UF t3) = UF(Node3 t1 b t2 a t3) |
node22 (Node3 t1 b t2 c t3) a (UF t4) = TD(Node2 (Node2 t1 b t2) c
(Node2 t3 a t4))
```

```
fun node31 :: 'a upD ⇒ 'a ⇒ 'a tree23 ⇒ 'a ⇒ 'a tree23 ⇒ 'a upD where
node31 (TD t1) a t2 b t3 = TD(Node3 t1 a t2 b t3) |
node31 (UF t1) a (Node2 t2 b t3) c t4 = TD(Node2 (Node3 t1 a t2 b t3)
c t4) |
node31 (UF t1) a (Node3 t2 b t3 c t4) d t5 = TD(Node3 (Node2 t1 a t2)
b (Node2 t3 c t4) d t5)
```

```
fun node32 :: 'a tree23 ⇒ 'a ⇒ 'a upD ⇒ 'a ⇒ 'a tree23 ⇒ 'a upD where
node32 t1 a (TD t2) b t3 = TD(Node3 t1 a t2 b t3) |
node32 t1 a (UF t2) b (Node2 t3 c t4) = TD(Node2 t1 a (Node3 t2 b t3 c
t4)) |
node32 t1 a (UF t2) b (Node3 t3 c t4 d t5) = TD(Node3 t1 a (Node2 t2 b
t3) c (Node2 t4 d t5))
```

```
fun node33 :: 'a tree23 ⇒ 'a ⇒ 'a tree23 ⇒ 'a ⇒ 'a upD ⇒ 'a upD where
node33 l a m b (TD r) = TD(Node3 l a m b r) |
node33 t1 a (Node2 t2 b t3) c (UF t4) = TD(Node2 t1 a (Node3 t2 b t3 c
t4)) |
node33 t1 a (Node3 t2 b t3 c t4) d (UF t5) = TD(Node3 t1 a (Node2 t2 b
t3) c (Node2 t4 d t5))
```

```
fun split_min :: 'a tree23 ⇒ 'a * 'a upD where
split_min (Node2 Leaf a Leaf) = (a, UF Leaf) |
split_min (Node3 Leaf a Leaf b Leaf) = (a, TD(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 completeness implies that so are the others. Exercise.

```
fun del :: 'a::linorder ⇒ 'a tree23 ⇒ 'a upD where
del x Leaf = TD Leaf |
del x (Node2 Leaf a Leaf) =
  (if x = a then UF Leaf else TD(Node2 Leaf a Leaf)) |
del x (Node3 Leaf a Leaf b Leaf) =
  TD(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',r') = split_min r in node22 l a' r') |
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 = *treeD*(*del x t*)

25.1 Functional Correctness

25.1.1 Proofs for isin

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

25.1.2 Proofs for insert

lemma *inorder_ins*:
  sorted(*inorder t*) ⟹ *inorder*(*treeI*(*ins x t*)) = *ins_list x* (*inorder t*)
by(*induction t*) (*auto simp*: *ins_list_simps split*: *upI.splits*)

lemma *inorder_insert*:

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

25.1.3 Proofs for delete

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

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

lemma *inorder_node31*: *height m* > *0* $\Longrightarrow$
inorder (*treeD* (*node31 l′ a m b r*)) = *inorder* (*treeD 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 (*treeD* (*node32 l a m′ b r*)) = *inorder l* @ *a* # *inorder* (*treeD m′*)
@ *b* # *inorder r*
by(*induct l a m′ b r rule*: *node32.induct*) *auto*

lemma *inorder_node33*: *height m* > *0* $\Longrightarrow$
inorder (*treeD* (*node33 l a m b r′*)) = *inorder l* @ *a* # *inorder m* @ *b* #
inorder (*treeD 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′*) $\Longrightarrow$ *complete t* $\Longrightarrow$ *height t* > *0* $\Longrightarrow$
x # *inorder*(*treeD t′*) = *inorder t*
by(*induction t arbitrary*: *t′ rule*: *split_min.induct*)
(*auto simp*: *inorder_nodes split*: *prod.splits*)

lemma *inorder_del*: ⟦ *complete t* ; *sorted*(*inorder t*) ⟧ $\Longrightarrow$
inorder(*treeD* (*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*: ⟦ *complete t* ; *sorted*(*inorder t*) ⟧ $\Longrightarrow$
inorder(*delete x t*) = *del_list x* (*inorder t*)

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

25.2 Completeness

25.2.1 Proofs for insert

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

fun *hI* :: *'a upI* ⇒ *nat* **where**
hI (*TI t*) = *height t* |
hI (*OF l a r*) = *height l*

lemma *complete_ins*: *complete t* ⟹ *complete* (*treeI*(*ins a t*)) ∧ *hI*(*ins a t*) = *height t*
by (*induct t*) (*auto split*!: *if_split upI.split*)

Now an alternative proof (by Brian Huffman) that runs faster because two properties (completeness 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* ⟷ *t* = *Leaf*
  by (*auto elim*: *full_0_elim intro*: *full.intros*)

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

lemma *full_Suc_Node2_iff* [*simp*]:
  full (*Suc n*) (*Node2 l p r*) ⟷ *full n l* ∧ *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*) ⟷ *full n l* ∧ *full n m* ∧ *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_complete*: *full n t* $\Longrightarrow$ *complete t*
  by (*induct set*: *full*, *auto dest*: *full_imp_height*)

lemma *complete_imp_full*: *complete t* $\Longrightarrow$ *full* (*height t*) *t*
  by (*induct t*, *simp_all*)

lemma *complete_iff_full*: *complete t* $\longleftrightarrow$ ($\exists$ *n*. *full n t*)
  by (*auto elim*!: *complete_imp_full full_imp_complete*)

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 *TI t* indicates that the height will be the same. A value of the form *OF l p r* indicates an increase in height.

fun $full_i$ :: *nat* $\Rightarrow$ *'a upI* $\Rightarrow$ *bool* **where**
$full_i$ *n* (*TI t*) $\longleftrightarrow$ *full n t* |
$full_i$ *n* (*OF 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*: *upI.split*)

The *insert* operation preserves completeance.

lemma *complete_insert*: *complete t* $\Longrightarrow$ *complete* (*insert a t*)
unfolding *complete_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

25.3 Proofs for delete

fun *hD* :: *'a upD* $\Rightarrow$ *nat* **where**
hD (*TD t*) = *height t* |
hD (*UF t*) = *height t* + *1*

lemma *complete_treeD_node21*:
  $[\![$*complete r*; *complete* (*treeD l'*); *height r* = *hD l'* $]\!]$ $\Longrightarrow$ *complete* (*treeD* (*node21 l' a r*))
by(*induct l' a r rule*: *node21.induct*) *auto*

lemma *complete_treeD_node22*:

$\llbracket$*complete*(*treeD r'*); *complete l*; *hD r'* = *height l* $\rrbracket \Longrightarrow$ *complete* (*treeD* (*node22 l a r'*))
by(*induct l a r' rule*: *node22.induct*) *auto*

lemma *complete_treeD_node31*:
$\llbracket$ *complete* (*treeD l'*); *complete m*; *complete r*; *hD l'* = *height r*; *height m* = *height r* $\rrbracket$
$\Longrightarrow$ *complete* (*treeD* (*node31 l' a m b r*))
by(*induct l' a m b r rule*: *node31.induct*) *auto*

lemma *complete_treeD_node32*:
$\llbracket$ *complete l*; *complete* (*treeD m'*); *complete r*; *height l* = *height r*; *hD m'* = *height r* $\rrbracket$
$\Longrightarrow$ *complete* (*treeD* (*node32 l a m' b r*))
by(*induct l a m' b r rule*: *node32.induct*) *auto*

lemma *complete_treeD_node33*:
$\llbracket$ *complete l*; *complete m*; *complete*(*treeD r'*); *height l* = *hD r'*; *height m* = *hD r'* $\rrbracket$
$\Longrightarrow$ *complete* (*treeD* (*node33 l a m b r'*))
by(*induct l a m b r' rule*: *node33.induct*) *auto*

lemmas *completes* = *complete_treeD_node21 complete_treeD_node22*
complete_treeD_node31 complete_treeD_node32 complete_treeD_node33

lemma *height'_node21*:
height r > *0* $\Longrightarrow$ *hD*(*node21 l' a r*) = *max* (*hD l'*) (*height r*) + *1*
by(*induct l' a r rule*: *node21.induct*)(*simp_all*)

lemma *height'_node22*:
height l > *0* $\Longrightarrow$ *hD*(*node22 l a r'*) = *max* (*height l*) (*hD r'*) + *1*
by(*induct l a r' rule*: *node22.induct*)(*simp_all*)

lemma *height'_node31*:
height m > *0* $\Longrightarrow$ *hD*(*node31 l a m b r*) =
max (*hD 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* $\Longrightarrow$ *hD*(*node32 l a m b r*) =
max (*height l*) (*max* (*hD 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 ⟹ hD(node33 l a m b r) =
max (height l) (max (height m) (hD 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 ⟹ complete t ⟹ hD t′ = height t
by(induct t arbitrary: x t′ rule: split_min.induct)
(auto simp: heights split: prod.splits)

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

lemma complete_split_min:
⟦ split_min t = (x, t′); complete t; height t > 0 ⟧ ⟹ complete (treeD t′)
by(induct t arbitrary: x t′ rule: split_min.induct)
(auto simp: heights height_split_min completes split: prod.splits)

lemma complete_treeD_del: complete t ⟹ complete(treeD(del x t))
by(induction x t rule: del.induct)
(auto simp: completes complete_split_min height_del height_split_min split: prod.splits)

corollary complete_delete: complete t ⟹ complete(delete x t)
by(simp add: delete_def complete_treeD_del)

25.4 Overall Correctness

interpretation S: Set_by_Ordered
where empty = empty **and** isin = isin **and** insert = insert **and** delete = delete
and inorder = inorder **and** inv = complete
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: complete_insert)

```
next
  case 7 thus ?case by(simp add: complete_delete)
qed (simp add: empty_def)+

end
```

26 2-3 Tree Implementation of Maps

```
theory Tree23_Map
imports
  Tree23_Set
  Map_Specs
begin

fun lookup :: ('a::linorder * 'b) tree23 ⇒ 'a ⇒ 'b option where
lookup Leaf x = None |
lookup (Node2 l (a,b) r) x = (case cmp x a of
  LT ⇒ 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))

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

GT ⇒ (*case cmp x* (*fst ab2*) *of*
LT ⇒ (*case upd x y m of*
TI m′ => *TI* (*Node3 l ab1 m′ ab2 r*)
| *OF m1 ab′ m2* => *OF* (*Node2 l ab1 m1*) *ab′* (*Node2 m2 ab2 r*)) |
EQ ⇒ *TI* (*Node3 l ab1 m* (*x*,*y*) *r*) |
GT ⇒ (*case upd x y r of*
TI r′ => *TI* (*Node3 l ab1 m ab2 r′*)
| *OF r1 ab′ r2* => *OF* (*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 = *treeI*(*upd a b t*)

fun *del* :: ′*a*::*linorder* ⇒ (′*a*∗′*b*) *tree23* ⇒ (′*a*∗′*b*) *upD* **where**
del x Leaf = *TD Leaf* |
del x (*Node2 Leaf ab1 Leaf*) = (*if x*=*fst ab1 then UF Leaf else TD*(*Node2 Leaf ab1 Leaf*)) |
del x (*Node3 Leaf ab1 Leaf ab2 Leaf*) = *TD*(*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 = *treeD*(*del x t*)

26.1 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*) $\Longrightarrow$ *inorder*(*treeI*(*upd x y t*)) = *upd_list x y* (*inorder t*)
by(*induction t*) (*auto simp*: *upd_list_simps split*: *upI.splits*)

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

lemma *inorder_del*: ⟦ *complete t* ; *sorted1*(*inorder t*) ⟧ $\Longrightarrow$
  inorder(*treeD* (*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*: ⟦ *complete t* ; *sorted1*(*inorder t*) ⟧ $\Longrightarrow$
  inorder(*delete x t*) = *del_list x* (*inorder t*)
by(*simp add*: *delete_def inorder_del*)

26.2 Balancedness

lemma *complete_upd*: *complete t* $\Longrightarrow$ *complete* (*treeI*(*upd x y t*)) ∧ *hI*(*upd x y t*) = *height t*
by (*induct t*) (*auto split*!: *if_split upI.split*)

corollary *complete_update*: *complete t* $\Longrightarrow$ *complete* (*update x y t*)
by (*simp add*: *update_def complete_upd*)

lemma *height_del*: *complete t* $\Longrightarrow$ *hD*(*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 *complete_treeD_del*: *complete t* $\Longrightarrow$ *complete*(*treeD*(*del x t*))
by(*induction x t rule*: *del.induct*)
  (*auto simp*: *completes complete_split_min height_del height_split_min split*: *prod.split*)

corollary *complete_delete*: *complete t* $\Longrightarrow$ *complete*(*delete x t*)
by(*simp add*: *delete_def complete_treeD_del*)

26.3 Overall Correctness

```
interpretation M: Map_by_Ordered
where empty = empty and lookup = lookup and update = update and
delete = delete
and inorder = inorder and inv = complete
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: complete_update)
next
  case 7 thus ?case by(simp add: complete_delete)
qed

end
```

27 2-3 Tree from List

```
theory Tree23_of_List
imports Tree23
begin
```

Linear-time bottom up conversion of a list of items into a complete 2-3 tree whose inorder traversal yields the list of items.

27.1 Code

Nonempty lists of 2-3 trees alternating with items, starting and ending with a 2-3 tree:

```
datatype 'a tree23s = T 'a tree23 | TTs 'a tree23 'a 'a tree23s
```

```
abbreviation not_T ts == (∀ t. ts ≠ T t)
```

```
fun len :: 'a tree23s ⇒ nat where
len (T _) = 1 |
len (TTs _ _ ts) = len ts + 1
```

fun *trees* :: *'a tree23s* ⇒ *'a tree23 set* **where**
trees (*T t*) = {*t*} |
trees (*TTs t a ts*) = {*t*} ∪ *trees ts*

Join pairs of adjacent trees:

fun *join_adj* :: *'a tree23s* ⇒ *'a tree23s* **where**
join_adj (*TTs t1 a* (*T t2*)) = *T*(*Node2 t1 a t2*) |
join_adj (*TTs t1 a* (*TTs t2 b* (*T t3*))) = *T*(*Node3 t1 a t2 b t3*) |
join_adj (*TTs t1 a* (*TTs t2 b ts*)) = *TTs* (*Node2 t1 a t2*) *b* (*join_adj ts*)

Towards termination of *join_all*:

lemma *len_ge2*:
  not_T ts ⟹ *len ts* ≥ *2*
by(*cases ts rule*: *join_adj.cases*) *auto*

lemma [*measure_function*]: *is_measure len*
by(*rule is_measure_trivial*)

lemma *len_join_adj_div2*:
  not_T ts ⟹ *len*(*join_adj ts*) ≤ *len ts div 2*
by(*induction ts rule*: *join_adj.induct*) *auto*

lemma *len_join_adj1*: *not_T ts* ⟹ *len*(*join_adj ts*) < *len ts*
using *len_join_adj_div2*[*of ts*] *len_ge2*[*of ts*] **by** *simp*

corollary *len_join_adj2*[*termination_simp*]: *len*(*join_adj* (*TTs t a ts*)) ≤ *len ts*
using *len_join_adj1*[*of TTs t a ts*] **by** *simp*

fun *join_all* :: *'a tree23s* ⇒ *'a tree23* **where**
join_all (*T t*) = *t* |
join_all ts = *join_all* (*join_adj ts*)

fun *leaves* :: *'a list* ⇒ *'a tree23s* **where**
leaves [] = *T Leaf* |
leaves (*a* # *as*) = *TTs Leaf a* (*leaves as*)

definition *tree23_of_list* :: *'a list* ⇒ *'a tree23* **where**
tree23_of_list as = *join_all*(*leaves as*)

27.2 Functional correctness

27.2.1 *inorder*:

fun *inorder2* :: *'a tree23s* ⇒ *'a list* **where**
inorder2 (*T t*) = *inorder t* |
inorder2 (*TTs t a ts*) = *inorder t* @ *a* # *inorder2 ts*

lemma *inorder2_join_adj*: *not_T ts* ⟹ *inorder2*(*join_adj ts*) = *inorder2 ts*
by (*induction ts rule*: *join_adj.induct*) *auto*

lemma *inorder_join_all*: *inorder* (*join_all ts*) = *inorder2 ts*
proof (*induction ts rule*: *join_all.induct*)
  case *1* **thus** *?case* **by** *simp*
next
  case (*2 t a ts*)
  thus *?case* **using** *inorder2_join_adj*[*of TTs t a ts*]
    by (*simp add*: *le_imp_less_Suc*)
qed

lemma *inorder2_leaves*: *inorder2*(*leaves as*) = *as*
by(*induction as*) *auto*

lemma *inorder*: *inorder*(*tree23_of_list as*) = *as*
by(*simp add*: *tree23_of_list_def inorder_join_all inorder2_leaves*)

27.2.2 Completeness:

lemma *complete_join_adj*:
  ∀ *t* ∈ *trees ts*. *complete t* ∧ *height t* = *n* ⟹ *not_T ts* ⟹
   ∀ *t* ∈ *trees* (*join_adj ts*). *complete t* ∧ *height t* = *Suc n*
by (*induction ts rule*: *join_adj.induct*) *auto*

lemma *complete_join_all*:
∀ *t* ∈ *trees ts*. *complete t* ∧ *height t* = *n* ⟹ *complete* (*join_all ts*)
proof (*induction ts arbitrary*: *n rule*: *join_all.induct*)
  case *1* **thus** *?case* **by** *simp*
next
  case (*2 t a ts*)
  thus *?case*
    apply *simp* **using** *complete_join_adj*[*of TTs t a ts n*, *simplified*] **by** *blast*
qed

lemma *complete_leaves*: $t \in$ *trees* (*leaves as*) $\Longrightarrow$ *complete t* $\wedge$ *height t* = *0*
by (*induction as*) *auto*

corollary *complete*: *complete*(*tree23_of_list as*)
by(*simp add*: *tree23_of_list_def complete_leaves complete_join_all*[*of _ 0*])

27.3 Linear running time

fun *T_join_adj* :: *'a tree23s* $\Rightarrow$ *nat* **where**
T_join_adj (*TTs t1 a* (*T t2*)) = *1* |
T_join_adj (*TTs t1 a* (*TTs t2 b* (*T t3*))) = *1* |
T_join_adj (*TTs t1 a* (*TTs t2 b ts*)) = *T_join_adj ts* + *1*

fun *T_join_all* :: *'a tree23s* $\Rightarrow$ *nat* **where**
T_join_all (*T t*) = *1* |
T_join_all ts = *T_join_adj ts* + *T_join_all* (*join_adj ts*) + *1*

fun *T_leaves* :: *'a list* $\Rightarrow$ *nat* **where**
T_leaves [] = *1* |
T_leaves (*a # as*) = *T_leaves as* + *1*

definition *T_tree23_of_list* :: *'a list* $\Rightarrow$ *nat* **where**
T_tree23_of_list as = *T_leaves as* + *T_join_all*(*leaves as*) + *1*

lemma *T_join_adj*: *not_T ts* $\Longrightarrow$ *T_join_adj ts* $\leq$ *len ts div 2*
by(*induction ts rule*: *T_join_adj.induct*) *auto*

lemma *len_ge_1*: *len ts* $\geq$ *1*
by(*cases ts*) *auto*

lemma *T_join_all*: *T_join_all ts* $\leq$ *2* $*$ *len ts*
proof(*induction ts rule*: *join_all.induct*)
  case *1* **thus** *?case* **by** *simp*
next
  case (*2 t a ts*)
  let *?ts* = *TTs t a ts*
  have *T_join_all ?ts* = *T_join_adj ?ts* + *T_join_all* (*join_adj ?ts*) + *1*
    by *simp*
  also have ... $\leq$ *len ?ts div 2* + *T_join_all* (*join_adj ?ts*) + *1*
    using *T_join_adj*[*of ?ts*] **by** *simp*
  also have ... $\leq$ *len ?ts div 2* + *2* $*$ *len* (*join_adj ?ts*) + *1*

```
    using 2.IH by simp
  also have ... ≤ len ?ts div 2 + 2 ∗ (len ?ts div 2) + 1
    using len_join_adj_div2[of ?ts] by simp
  also have ... ≤ 2 ∗ len ?ts using len_ge_1[of ?ts] by linarith
  finally show ?case .
qed

lemma T_leaves: T_leaves as = length as + 1
by(induction as) auto

lemma len_leaves: len(leaves as) = length as + 1
by(induction as) auto

lemma T_tree23_of_list: T_tree23_of_list as ≤ 3∗(length as) + 4
using T_join_all[of leaves as] by(simp add: T_tree23_of_list_def T_leaves
len_leaves)

end
```

28 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

29 2-3-4 Tree Implementation of Sets

theory *Tree234_Set*
imports
  Tree234
  Cmp
  Set_Specs
begin

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

29.1 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)
```

29.2 Functional correctness

29.2.1 Functional correctness of isin:

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

29.2.2 Functional correctness of insert:

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

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

29.2.3 Functional correctness of 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$(induct\ l\ a\ m\ b\ r'\ rule: node33.induct)\ auto$

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*) $]\!]$ $\Longrightarrow$
  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*) $]\!]$ $\Longrightarrow$
  inorder(*delete x t*) = *del_list x* (*inorder t*)
by(*simp add*: *delete_def inorder_del*)

29.3 Balancedness

29.3.1 Proofs for insert

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

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

fun $\mathit{height\_up_i}$ :: $'a\ \mathit{up_i} \Rightarrow \mathit{nat}$ **where**
$\mathit{height}\ (T_i\ t) = \mathit{height}\ t$ |
$\mathit{height}\ (\mathit{Up_i}\ l\ a\ r) = \mathit{height}\ l$

instance ..

end

lemma *bal_ins*: $\mathit{bal}\ t \Longrightarrow \mathit{bal}\ (\mathit{tree_i}(\mathit{ins}\ a\ t)) \wedge \mathit{height}(\mathit{ins}\ a\ t) = \mathit{height}\ t$
by (*induct t*) (*auto split*!: *if_split* $\mathit{up_i}$*.split*)

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

inductive *full* :: $\mathit{nat} \Rightarrow 'a\ \mathit{tree234} \Rightarrow \mathit{bool}$ **where**
full 0 Leaf |
$[\![\mathit{full}\ n\ l;\ \mathit{full}\ n\ r]\!] \Longrightarrow \mathit{full}\ (\mathit{Suc}\ n)\ (\mathit{Node2}\ l\ p\ r)$ |
$[\![\mathit{full}\ n\ l;\ \mathit{full}\ n\ m;\ \mathit{full}\ n\ r]\!] \Longrightarrow \mathit{full}\ (\mathit{Suc}\ n)\ (\mathit{Node3}\ l\ p\ m\ q\ r)$ |
$[\![\mathit{full}\ n\ l;\ \mathit{full}\ n\ m;\ \mathit{full}\ n\ m';\ \mathit{full}\ n\ r]\!] \Longrightarrow \mathit{full}\ (\mathit{Suc}\ n)\ (\mathit{Node4}\ l\ p\ m\ q\ m'\ q'$
$r)$

inductive_cases *full_elims*:
  full n Leaf
  $\mathit{full}\ n\ (\mathit{Node2}\ l\ p\ r)$
  $\mathit{full}\ n\ (\mathit{Node3}\ l\ p\ m\ q\ r)$
  $\mathit{full}\ n\ (\mathit{Node4}\ l\ p\ m\ q\ m'\ q'\ r)$

inductive_cases *full_0_elim*: *full 0 t*
inductive_cases *full_Suc_elim*: $\mathit{full}\ (\mathit{Suc}\ n)\ t$

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

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

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

lemma *full_Suc_Node3_iff* [*simp*]:
  $\mathit{full}\ (\mathit{Suc}\ n)\ (\mathit{Node3}\ l\ p\ m\ q\ r) \longleftrightarrow \mathit{full}\ n\ l \wedge \mathit{full}\ n\ m \wedge \mathit{full}\ n\ r$
  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

29.3.2 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*:
  $[\![\ bal\ (tree_d\ l);\ bal\ m;\ bal\ n;\ bal\ r;\ height\ l = height\ r;\ height\ m = height$
$r;\ height\ n = height\ r\ ]\!]$
  $\Longrightarrow bal\ (tree_d\ (node41\ l\ a\ m\ b\ n\ c\ r))$
by(*induct l a m b n c r rule*: *node41.induct*) *auto*

lemma *bal_tree$_d$_node42*:
  $[\![\ bal\ l;\ bal\ (tree_d\ m);\ bal\ n;\ bal\ r;\ height\ l = height\ r;\ height\ m = height$
$r;\ height\ n = height\ r\ ]\!]$
  $\Longrightarrow bal\ (tree_d\ (node42\ l\ a\ m\ b\ n\ c\ r))$
by(*induct l a m b n c r rule*: *node42.induct*) *auto*

lemma *bal_tree$_d$_node43*:
  $[\![\ bal\ l;\ bal\ m;\ bal\ (tree_d\ n);\ bal\ r;\ height\ l = height\ r;\ height\ m = height$
$r;\ height\ n = height\ r\ ]\!]$

$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *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* ⟹ *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* ⟹ *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'*) ⟹ *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 add*: *heights height_split_min split*!: *if_split prod.split*)

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*!: *if_split prod.split*)

corollary *bal_delete*: *bal t* ⟹ *bal*(*delete x t*)
by(*simp add*: *delete_def bal_*$tree_d$*_del*)

29.4 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
```

30 2-3-4 Tree Implementation of Maps

```
theory Tree234_Map
imports
  Tree234_Set
  Map_Specs
begin
```

30.1 Map operations on 2-3-4 trees

```
fun lookup :: ('a::linorder * 'b) tree234 ⇒ 'a ⇒ 'b option where
lookup Leaf x = None |
lookup (Node2 l (a,b) r) x = (case cmp x a of
  LT ⇒ 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) ab24 (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
                    Ti t4' => Ti (Node4 t1 ab1 t2 ab2 t3 ab3 t4')
                  | Upi t41 q t42 => Upi (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) upd where
del x Leaf = Td Leaf |
del x (Node2 Leaf ab1 Leaf) = (if x=fst ab1 then Upd Leaf else Td(Node2
Leaf ab1 Leaf)) |
del x (Node3 Leaf ab1 Leaf ab2 Leaf) = Td(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) =
  Td(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*)

30.2 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*)

30.3 Balancedness

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

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

lemma *height_del*: *bal t* ⟹ *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* ⟹ *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*)

30.4 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

31 1-2 Brother Tree Implementation of Sets

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

31.1 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
```

31.2 Invariants

```
fun B :: nat ⇒ 'a bro set
and U :: nat ⇒ 'a bro set where
B 0 = {N0} |
B (Suc h) = { N2 t1 a t2 | t1 a t2.
  t1 ∈ B h ∪ U h ∧ t2 ∈ B h ∨ t1 ∈ B h ∧ t2 ∈ B h ∪ U h} |
U 0 = {} |
U (Suc h) = N1 ` B h

abbreviation T h ≡ B h ∪ U h

fun Bp :: nat ⇒ 'a bro set where
Bp 0 = B 0 ∪ L2 ` UNIV |
Bp (Suc 0) = B (Suc 0) ∪ {N3 N0 a N0 b N0|a b. True} |
Bp (Suc(Suc h)) = B (Suc(Suc h)) ∪
  {N3 t1 a t2 b t3 | t1 a t2 b t3. t1 ∈ B (Suc h) ∧ t2 ∈ U (Suc h) ∧ t3 ∈
B (Suc h)}

fun Um :: nat ⇒ 'a bro set where
Um 0 = {} |
Um (Suc h) = N1 ` T h
```

31.3 Functional Correctness Proofs

31.3.1 Proofs for isin

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

31.3.2 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*: $t \in T\ h \Longrightarrow$
  $sorted(inorder\ t) \Longrightarrow inorder(ins\ a\ t) = ins\_list\ a\ (inorder\ t)$
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

31.3.3 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*)
  apply (*induction h arbitrary*: *t*)
  apply (*auto simp*: *del_list_simps inorder_n2 split*: *option.splits*)
  apply (*auto simp*: *del_list_simps inorder_n2*
      inorder_split_min[*OF UnI1*] *inorder_split_min*[*OF UnI2*] *split*: *option.splits*)
  done

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

31.4 Invariant Proofs

31.4.1 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*:
  $(t1 \in Bp\ h \wedge t2 \in T\ h \longrightarrow n2\ t1\ a\ t2 \in Bp\ (Suc\ h)) \wedge$
   $(t1 \in T\ h \wedge t2 \in Bp\ h \longrightarrow n2\ t1\ a\ t2 \in Bp\ (Suc\ h))$
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*: $t \in B\ h \Longrightarrow t \in Bp\ h$
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
```

31.4.2 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 ∈ Um h; t2 ∈ B h⟧ ⟹ n2 t1 a t2 ∈ 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 ∈ B h ; t2 ∈ Um h ⟧ ⟹ n2 t1 a t2 ∈ 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 ∈ T h ; t2 ∈ T h ⟧ ⟹ n2 t1 a t2 ∈ 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 ∈ B h; split_min t = None⟧ ⟹  t = N0
by (cases t) (auto split: option.splits)

lemma split_minNoneN1 : ⟦t ∈ U h; split_min t = None⟧ ⟹ t = N1 N0
by (cases h) (auto simp: split_minNoneN0  split: option.splits)

lemma split_min_type:
  t ∈ B h ⟹ split_min t = Some (a, t′) ⟹ t′ ∈ T h
  t ∈ U h ⟹ split_min t = Some (a, t′) ⟹ t′ ∈ 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 ∈ T h t2 ∈ T h t1 ∈ B h ∨ t2 ∈ 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*: $t \in T\ (h{+}1) \Longrightarrow$ *tree* $t \in B\ (h{+}1) \cup B\ h$
by(*auto*)

lemma *delete_type*: $t \in B\ h \Longrightarrow$ *delete* $x\ t \in B\ h \cup B(h{-}1)$
unfolding *delete_def*
by (*cases h*) (*simp*, *metis del_type*(*1*) *tree_type Suc_eq_plus1 diff_Suc_1*)

end

31.5 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* = $\lambda t.\ \exists h.\ t \in$
$B\ h$
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*)

31.6 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

32 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
```

32.1 Functional Correctness Proofs

32.1.1 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)
```

32.1.2 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 ∈ T h ⟹
  sorted1(inorder t) ⟹ inorder(update x y t) = upd_list x y (inorder t)
by(simp add: update_def inorder_upd inorder_tree)

end
```

32.1.3 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*)
  apply (*induction h arbitrary*: *t*)
  apply (*auto simp*: *del_list_simps inorder_n2*)
  apply (*auto simp*: *del_list_simps inorder_n2*
      inorder_split_min[*OF UnI1*] *inorder_split_min*[*OF UnI2*] *split*: *option.splits*)
  done

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

32.2 Invariant Proofs

32.2.1 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

32.2.2 Proofs for deletion

context *delete*
begin

lemma *del_type*:
  $t \in B\ h \Longrightarrow$ *del x t* $\in T\ h$

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

32.3 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
```

33 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]*

33.1 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*+

33.2 Invariance

33.2.1 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 rule*: *tree2_cases*)
      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* $\Longrightarrow$ *skew t = t*
by(*cases t rule*: *skew.cases*) *auto*

lemma *split_invar*: *invar t* $\Longrightarrow$ *split t = t*
by(*cases t rule*: *split.cases*) *clarsimp+*

lemma *invar_NodeL*:
  $\llbracket$ *invar*(*Node l* (*x*, *n*) *r*); *invar l'*; *lvl l' = lvl l* $\rrbracket \Longrightarrow$ *invar*(*Node l'* (*x*, *n*) *r*)
by(*auto*)

lemma *invar_NodeR*:
  $\llbracket$ *invar*(*Node l* (*x*, *n*) *r*); *n = lvl r + 1*; *invar r'*; *lvl r' = lvl r* $\rrbracket \Longrightarrow$ *invar*(*Node l* (*x*, *n*) *r'*)
by(*auto*)

lemma *invar_NodeR2*:
  $\llbracket$ *invar*(*Node l* (*x*, *n*) *r*); *sngl r'*; *n = lvl r + 1*; *invar r'*; *lvl r' = n* $\rrbracket \Longrightarrow$ *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*) $\longleftrightarrow$
  ($\exists$ *l x r*. *insert a t = Node l* (*x*, *lvl t + 1*) *r* $\wedge$ *lvl l = lvl r*)
apply(*cases t rule*: *tree2_cases*)
apply(*auto simp add*: *skew_case split_case split*: *if_splits*)
apply(*auto split*: *tree.splits if_splits*)
done

lemma *invar_insert*: *invar t* $\Longrightarrow$ *invar*(*insert a t*)
proof(*induction t rule*: *tree2_induct*)
  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 rule: tree2_cases)
      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
```

33.2.2 Proofs for delete

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

lemma invarR: ASSUMPTION(invar ⟨l, (a,lv), 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 rule: tree2_cases) (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 in-
var.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 rule: tree2_cases) 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⟨l, (a,lv), r⟩ pre_adjust ⟨l, (a,lv), r′⟩ post_del r r′
shows post_del ⟨l, (a,lv), r⟩ (adjust ⟨l, (a,lv), r′⟩)
proof(unfold post_del_def, safe del: disjCI)
  let ?t = ⟨l, (a,lv), r⟩
  let ?t′ = adjust ⟨l, (a,lv), 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 ⟨l, (a,lv), r'⟩
    proof(cases r' rule: tree2_cases)
      case Leaf thus ?thesis by simp
    next
      case Node thus ?thesis using as(2) assms(1,3)
      by (cases r rule: tree2_cases) (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 l a lv rl bl rr)
  let ?r =  ⟨rl, bl, rr⟩
  let ?t = ⟨l, (a, lv), ?r⟩
  from 2.prems(2) obtain r' where r': (r', x) = split_max ?r
    and [simp]: t' = adjust ⟨l, (a, lv), 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 rule: tree2_induct)
  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*]

33.3 Functional Correctness

33.3.1 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*)

33.3.2 Proofs for delete

lemma *inorder_adjust*: *t* ≠ *Leaf* ⟹ *pre_adjust t* ⟹ *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 ≠ Leaf; invar t ⟧ ⟹ 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 ⟹ sorted(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 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
```

34 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)))
```

34.1 Invariance

34.1.1 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 rule: tree2_induct)
  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 rule: tree2_cases)
      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
```

34.1.2 Proofs for delete

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

theorem post_delete: invar t ⟹ post_del t (delete x t)
proof (induction t rule: tree2_induct)
  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)
```

34.2 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

35 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* (*Node l* (*a*, *b*) *r*) ⟹ *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*,*b*) *r*) ⟧ ⟹ *inv l* ∧ *inv r*
begin

declare *set_join* [*simp*] *Let_def*[*simp*]

35.1 *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* = {*m*} ∪ *set_tree t'*
proof(*induction t arbitrary*: *t' rule*: *tree2_induct*)
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' rule*: *tree2_induct*)
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' rule*: *tree2_induct*)
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

35.2 *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*)

35.3 *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 =
  (case cmp x a of
     LT ⇒ let (l1,b,l2) = split l x in (l1, b, join l2 a r) |
     GT ⇒ let (r1,b,r2) = split r x in (join l a r1, b, r2) |
     EQ ⇒ (l, True, r))

lemma split: split t x = (l,b,r) ⟹ bst t ⟹
  set_tree l = {a ∈ set_tree t. a < x} ∧ set_tree r = {a ∈ set_tree t. x <
a}
  ∧ (b = (x ∈ set_tree t)) ∧ bst l ∧ bst r
proof(induction t arbitrary: l b r rule: tree2_induct)
  case Leaf thus ?case by simp
next
  case (Node y a b z l c r)
  consider (LT) l1 xin l2 where (l1,xin,l2) = split y x
    and split ⟨y, (a, b), z⟩ x = (l1, xin, join l2 a z) and cmp x a = LT
  | (GT) r1 xin r2 where (r1,xin,r2) = split z x
    and split ⟨y, (a, b), z⟩ x = (join y a r1, xin, r2) and cmp x a = GT
  | (EQ) split ⟨y, (a, b), z⟩ x = (y, True, z) and cmp x a = EQ
    by (force split: cmp_val.splits prod.splits if_splits)

  thus ?case
  proof cases
    case (LT l1 xin l2)
    with Node.IH(1)[OF ‹(l1,xin,l2) = split y x›[symmetric]] Node.prems
    show ?thesis by (force intro!: bst_join)
  next
    case (GT r1 xin r2)
    with Node.IH(2)[OF ‹(r1,xin,r2) = split z x›[symmetric]] Node.prems
    show ?thesis by (force intro!: bst_join)
  next
    case EQ
    with Node.prems show ?thesis by auto
  qed
qed

lemma split_inv: split t x = (l,b,r) ⟹ inv t ⟹ inv l ∧ inv r
proof(induction t arbitrary: l b r rule: tree2_induct)
  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]
```

35.4 *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) = {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 ⟹ inv (insert x t)
by(force simp: insert_def inv_join dest: split_inv split: prod.split)
```

35.5 *delete*

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

lemma set_tree_delete: bst t ⟹ set_tree (delete x t) = set_tree t − {x}
by(auto simp: delete_def split split: prod.split)

lemma bst_delete: bst t ⟹ bst (delete x t)
by(force simp add: delete_def intro: bst_join2 dest: split split: prod.split)

lemma inv_delete: inv t ⟹ inv (delete x t)
by(force simp: delete_def inv_join2 dest: split_inv split: prod.split)
```

35.6 *union*

```
fun union :: ('a*'b)tree ⇒ ('a*'b)tree ⇒ ('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 ⇒
   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 ⟹ 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 ⟧ ⟹ 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
```

35.7 *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,b,r2) = split t2 a;
       l' = inter l1 l2; r' = inter r1 r2
   in if b 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 rule: tree2_cases)
    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 b r2 where sp: split t2 a = (l2,b,r2) using prod_cases3 by
blast
      let ?L2 = set_tree l2 let ?R2 = set_tree r2 let ?A = if b then {a}
else {}
      have t2: set_tree t2 = ?L2 ∪ ?R2 ∪ ?A 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
∪ ?A)
        by(simp add: t2)
      also have ... = (?L1 ∩ ?L2) ∪ (?R1 ∩ ?R2) ∪ ?A
        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
        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
        split!: tree.split prod.split dest: inv_Node)
qed
```

35.8 *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 rule: tree2_cases)
    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 b r1 where sp: split t1 a = (l1,b,r1) using prod_cases3 by
blast
      let ?L1 = set_tree l1 let ?R1 = set_tree r1 let ?A = if b then {a}
else {}
      have t1: set_tree t1 = ?L1 ∪ ?R1 ∪ ?A and
```

```
          **: $a \notin$ ?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
        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
        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 = λl 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
```

36 Join-Based Implementation of Sets via RBTs

```
theory Set2_Join_RBT
imports
  Set2_Join
  RBT_Set
begin
```

36.1 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 ⇒ 'a ⇒ 'a rbt ⇒ '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))

definition 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]
```

36.2 Properties

36.2.1 Color and height invariants

```
lemma invc2_joinL:
 ⟦ invc l; invc r; bheight l ≤ bheight r ⟧ ⟹
  invc2 (joinL l x r)
```

$\wedge$ (*bheight l* $\neq$ *bheight r* $\wedge$ *color r* = *Black* $\longrightarrow$ *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* $\geq$ *bheight r* ⟧ $\Longrightarrow$
invc2 (*joinR l x r*)
$\wedge$ (*bheight l* $\neq$ *bheight r* $\wedge$ *color l* = *Black* $\longrightarrow$ *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* $\leq$ *bheight r* ⟧ $\Longrightarrow$ *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* $\leq$ *bheight r* ⟧ $\Longrightarrow$ *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_joinR*:
⟦ *invh l*; *invh r*; *bheight l* $\geq$ *bheight r* ⟧ $\Longrightarrow$ *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* $\geq$ *bheight r* ⟧ $\Longrightarrow$ *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

All invariants in one:

lemma *inv_joinL*: ⟦ *invc l*; *invc r*; *invh l*; *invh r*; *bheight l* ≤ *bheight r* ⟧
⟹ *invc2* (*joinL l x r*) ∧ (*bheight l* ≠ *bheight r* ∧ *color r* = *Black* ⟶
invc (*joinL l x r*))
∧ *invh* (*joinL l x 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*: *inv_baliL invc2I joinL.simps*[*of l x r*] *split*!: *tree.splits*
if_splits)
qed

lemma *inv_joinR*: ⟦ *invc l*; *invc r*; *invh l*; *invh r*; *bheight l* ≥ *bheight r* ⟧
⟹ *invc2* (*joinR l x r*) ∧ (*bheight l* ≠ *bheight r* ∧ *color l* = *Black* ⟶
invc (*joinR l x r*))
∧ *invh* (*joinR l x r*) ∧ *bheight* (*joinR l x r*) = *bheight l*
proof (*induct l x r rule*: *joinR.induct*)
case (*1 l x r*) **thus** *?case*
by(*auto simp*: *inv_baliR invc2I joinR.simps*[*of l x r*] *split*!: *tree.splits*
if_splits)
qed

lemma *rbt_join*: ⟦ *invc l*; *invh l*; *invc r*; *invh r* ⟧ ⟹ *rbt*(*join l x r*)
by(*simp add*: *inv_joinL inv_joinR invh_paint rbt_def color_paint_Black*
join_def)

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 join_def*)

36.2.2 Inorder properties

Currently unused. Instead *Tree2.set_tree* and *Tree2.bst* properties are proved directly.

lemma *inorder_joinL*: *bheight l* $\leq$ *bheight r* $\Longrightarrow$ *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 join_def*
      split!: *tree.splits color.splits if_splits*
      dest!: *arg_cong*[**where** *f* = *inorder*])

36.2.3 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 $\leq$ *bheight r* $\Longrightarrow$ *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 join_def)

lemma bst_baliL:
  ⟦bst l; bst r; ∀x∈set_tree l. x < a; ∀x∈set_tree r. a < x⟧
   ⟹ bst (baliL l a r)
by(cases (l,a,r) rule: baliL.cases) (auto simp: ball_Un)

lemma bst_baliR:
  ⟦bst l; bst r; ∀x∈set_tree l. x < a; ∀x∈set_tree r. a < x⟧
   ⟹ bst (baliR l a r)
by(cases (l,a,r) rule: baliR.cases) (auto simp: ball_Un)

lemma bst_joinL:
  ⟦bst (Node l (a, n) r); bheight l ≤ bheight r⟧
  ⟹ bst (joinL l a r)
proof(induction l a r rule: joinL.induct)
  case (1 l a r)
  thus ?case
    by(auto simp: set_baliL joinL.simps[of l a 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 < a; ∀y∈set_tree r. a < y ⟧
  ⟹ bst (joinR l a r)
proof(induction l a r rule: joinR.induct)
  case (1 l a r)
  thus ?case
   by(auto simp: set_baliR joinR.simps[of l a 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 (*Node l* (*a*, *n*) *r*) $\Longrightarrow$ *bst* (*join l a r*)
by(*auto simp*: *bst_paint bst_joinL bst_joinR join_def*)

lemma *inv_join*: ⟦ *invc l*; *invh l*; *invc r*; *invh r* ⟧ $\Longrightarrow$ *invc*(*join l x r*) $\wedge$ *invh*(*join l x r*)
by (*simp add*: *inv_joinL inv_joinR invh_paint join_def*)

36.2.4 Interpretation of *Set2_Join* with Red-Black Tree

global_interpretation *RBT*: *Set2_Join*
where *join* = *join* **and** *inv* = λ*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** (*simp add*: *bst_join*)
next
  case *3* **show** *?case* **by** *simp*
next
  case *4* **thus** *?case* **by** (*simp add*: *inv_join*)
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* ⇒ *nat* ⇒ *'a*
fixes *update* :: *nat* ⇒ *'a* ⇒ *'ar* ⇒ *'ar*
fixes *len* :: *'ar* ⇒ *nat*

fixes *array* :: *'a list* ⇒ *'ar*

fixes *list* :: *'ar* ⇒ *'a list*
fixes *invar* :: *'ar* ⇒ *bool*

assumes *lookup*: *invar ar* ⟹ *n* < *len ar* ⟹ *lookup ar n* = *list ar* ! *n*
assumes *update*: *invar ar* ⟹ *n* < *len ar* ⟹ *list*(*update n x ar*) = (*list ar*)[*n*:=*x*]
assumes *len_array*: *invar ar* ⟹ *len ar* = *length* (*list ar*)
assumes *array*: *list* (*array xs*) = *xs*

assumes *invar_update*: *invar ar* ⟹ *n* < *len ar* ⟹ *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

37 Braun Trees

theory *Braun_Tree*
imports *HOL−Library.Tree_Real*
begin

Braun Trees were studied by Braun and Rem [5] and later Hoogerwoord [10].

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 almost complete:

lemma *acomplete_if_braun*: *braun t* ⟹ *acomplete t*
proof(*induction t*)
  case *Leaf* **show** *?case* **by** (*simp add*: *acomplete_def*)
next
  case (*Node l x r*) **thus** *?case* **using** *acomplete_Node_if_wbal2* **by** *force*
qed

37.1 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*

One direction:

```
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
```

The other direction is more complicated. The following proof is due to Thomas Sewell.

```
lemma disj_evens_odds: (∗) 2 ‘ A ∩ Suc ‘ (∗) 2 ‘ B = {}
using double_not_eq_Suc_double by auto

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 braun_indices_intvl_base_1:
  assumes bi: braun_indices t = {m..n}
  shows {m..n} = {1..size t}
proof (cases t = Leaf)
  case True then show ?thesis using bi by simp
next
  case False
  note eqs = eqset_imp_iff[OF bi]
  from eqs[of 0] have 0: 0 < m
    by (simp add: braun_indices1)
  from eqs[of 1] have 1: m ≤ 1
    by (cases t; simp add: False)
  from 0 1 have eq1: m = 1 by simp
  from card_braun_indices[of t] show ?thesis
    by (simp add: bi eq1)
qed

lemma even_of_intvl_intvl:
  fixes S :: nat set
  assumes S = {m..n} ∩ {i. even i}
  shows ∃ m' n'. S = (λi. i * 2) ‘ {m'..n'}
  apply (rule exI[where x=Suc m div 2], rule exI[where x=n div 2])
  apply (fastforce simp add: assms mult.commute)
  done

lemma odd_of_intvl_intvl:
  fixes S :: nat set
  assumes S = {m..n} ∩ {i. odd i}
  shows ∃ m' n'. S = Suc ‘ (λi. i * 2) ‘ {m'..n'}
proof –
  have step1: ∃ m'. S = Suc ‘ ({m'..n − 1} ∩ {i. even i})
    apply (rule_tac x=if n = 0 then 1 else m − 1 in exI)
    apply (auto simp: assms image_def elim!: oddE)
    done
  thus ?thesis
    by (metis even_of_intvl_intvl)
qed

lemma image_int_eq_image:
  (∀ i ∈ S. f i ∈ T) ⟹ (f ‘ S) ∩ T = f ‘ S
  (∀ i ∈ S. f i ∉ T) ⟹ (f ‘ S) ∩ T = {}
```

  by *auto*

lemma *braun_indices1_le*:
  i ∈ *braun_indices t* ⟹ *Suc 0* ≤ *i*
  using *braun_indices1 not_less_eq_eq* **by** *blast*

lemma *braun_if_braun_indices*: *braun_indices t* = {*1..size t*} ⟹ *braun t*
proof(*induction t*)
case *Leaf*
  then show *?case* **by** *simp*
next
  case (*Node l x r*)
  obtain *t* **where** *t*: *t* = *Node l x r* **by** *simp*
  from *Node.prems* **have** *eq*: {*2* .. *size t*} = (λ*i*. *i* ∗ *2*) ‘ *braun_indices l* ∪ *Suc* ‘ (λ*i*. *i* ∗ *2*) ‘ *braun_indices r*
    (**is** *?R* = *?S* ∪ *?T*)
    apply *clarsimp*
    apply (*drule_tac f*=λ*S*. *S* ∩ {*2..*} **in** *arg_cong*)
   apply (*simp add*: *t mult.commute Int_Un_distrib2 image_int_eq_image braun_indices1_le*)
    done
  then have *ST*: *?S* = *?R* ∩ {*i*. *even i*} *?T* = *?R* ∩ {*i*. *odd i*}
    by (*simp_all add*: *Int_Un_distrib2 image_int_eq_image*)
  from *ST* **have** *l*: *braun_indices l* = {*1* .. *size l*}
   by (*fastforce dest*: *braun_indices_intvl_base_1 dest*!: *even_of_intvl_intvl*
                simp: *mult.commute inj_image_eq_iff*[*OF inj_onI*])
  from *ST* **have** *r*: *braun_indices r* = {*1* .. *size r*}
   by (*fastforce dest*: *braun_indices_intvl_base_1 dest*!: *odd_of_intvl_intvl*
                simp: *mult.commute inj_image_eq_iff*[*OF inj_onI*])
  note *STa* = *ST*[*THEN eqset_imp_iff*, *THEN iffD2*]
  note *STb* = *STa*[*of size t*] *STa*[*of size t* − *1*]
  then have *sizes*: *size l* = *size r* ∨ *size l* = *size r* + *1*
    apply (*clarsimp simp*: *t l r inj_image_mem_iff*[*OF inj_onI*])
     apply (*cases even* (*size l*); *cases even* (*size r*); *clarsimp elim*!: *oddE*; *fastforce*)
    done
  from *l r sizes* **show** *?case*
    by (*clarsimp simp*: *Node.IH*)
qed

lemma *braun_iff_braun_indices*: *braun t* ⟷ *braun_indices t* = {*1..size t*}
using *braun_if_braun_indices braun_indices_if_braun* **by** *blast*

end

38 Arrays via Braun Trees

theory *Array_Braun*
imports
  Array_Specs
  Braun_Tree
begin

38.1 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)

38.1.1 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*: $\llbracket$ *braun t*; *n* $\in$ {*1*.. *size t*} $\rrbracket \Longrightarrow$
  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*: $\llbracket$ *odd n*; *odd m* $\rrbracket \Longrightarrow$ *n div 2* = (*m*::*nat*) *div 2* $\longleftrightarrow$ *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*: $\llbracket$ *braun t*; *n* $\in$ {*1*.. *size t*} $\rrbracket \Longrightarrow$ *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 $\llbracket$*braun ?t*; *?n* $\in$ {*1*..*size ?t*}$\rrbracket \Longrightarrow$ *list* (*update1 ?n ?x ?t*) = (*list ?t*)[*?n* − *1* := *?x*]:

lemma *diff1_eq_iff*: *n* > *0* $\Longrightarrow$ *n* − *Suc 0* = *m* $\longleftrightarrow$ *n* = *m*+*1*
by *arith*

lemma *list_update_splice*:
  $\llbracket$ *n* < *size xs* + *size ys*; *size ys* ≤ *size xs*; *size xs* ≤ *size ys* + *1* $\rrbracket \Longrightarrow$
  (*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*: $\llbracket$ *braun t*; *n* $\in$ {*1*.. *size t*} $\rrbracket \Longrightarrow$ *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* $\Longrightarrow$ *splice* (*xs* @ [*x*]) *ys* = *splice xs ys* @ [*x*]
and *size ys*+*1* ≥ *size xs* $\Longrightarrow$ *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* $\Longrightarrow$ *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* $\Longrightarrow$ *m* = *size t* $\Longrightarrow$ *size*(*update1* (*Suc m*) *x t*) = *size t* + *1*
by(*induction t arbitrary*: *m*)(*auto*)

lemma *braun_add_hi*: *braun t* $\Longrightarrow$ *braun*(*update1* (*Suc*(*size t*)) *x t*)
by(*induction t*)(*auto simp*: *size_add_hi*)

lemma *size_braun_adds*:
 $[\![$ *braun t*; *size t* = *n* $]\!]$ $\Longrightarrow$ *size*(*adds xs n t*) = *size t* + *length xs* $\wedge$ *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* $]\!]$ $\Longrightarrow$ *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*)

38.1.2 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* $\wedge$ *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

38.2 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*))

38.2.1 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*)

38.2.2 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

38.3 Faster

38.3.1 Size

fun *diff* :: *'a tree* ⇒ *nat* ⇒ *nat* **where**
diff Leaf _ = 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*)

declare *Let_def*[*simp*]

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*: *diff*)

38.3.2 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*)

38.3.3 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*)

This is the more concise definition but seems to complicate the proofs:

lemma *take_nths_eq_nths*: *take_nths i k xs* = *nths xs* (⋃*n*. {*n*∗*2^k* + *i*})
proof(*induction xs arbitrary*: *i*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons x xs*)
  show *?case*

```
  proof cases
    assume [simp]: i = 0
    have (⋃ n. {(n+1) * 2 ^ k - 1}) = {m. ∃ n. Suc m = n * 2 ^ k}
      apply (auto simp del: mult_Suc)
      by (metis diff_Suc_Suc diff_zero mult_eq_0_iff not0_implies_Suc)
    thus ?thesis by (simp add: Cons.IH ac_simps nths_Cons)
  next
    assume [arith]: i ≠ 0
    have (⋃ n. {n * 2 ^ k + i - 1}) = {m. ∃ n. Suc m = n * 2 ^ k + i}
      apply auto
      by (metis diff_Suc_Suc diff_zero)
    thus ?thesis by (simp add: Cons.IH nths_Cons)
  qed
qed

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
```

38.3.4 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] 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)
  show ?case
    using less.prems
    by (auto simp: brauns.simps[of k xs] nth_nodes take_nths_take_nths
                    IH take_nths_empty hd_take_nths length_brauns)
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)
  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)
    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
```

38.3.5 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 (λt. 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:
      ∃ n. ts = map (λx. 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)
  next
    case False
    with less.prems(2) have *:
      ∀ i < 2 ^ k. ts ! i ≠ Leaf
         ∧ value (ts ! i) = xs ! i
         ∧ braun_list (left (ts ! i)) (take_nths (i + 2 ^ k) (Suc k) xs)
         ∧ (∀ ys. ys = take_nths (i + 2 ∗ 2 ^ k) (Suc k) xs
                 ⟶ 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 * 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 ⟹
  (∑ t←ts. k * size t) = (∑ t ← 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 (λt. 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 (λt. t ≠ Leaf) ts
  show ?case
  proof cases
    assume ?us = []
    thus ?thesis using T_list_fast_rec.simps[of ts]
      by(simp add: 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)
    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 ... = ($\sum$ *t*←*?us*. *7*∗*size t*) + *length ts*
      by(*simp add*: *sum_tree_list_children*)
    also have ... ≤ ($\sum$ *t*←*ts*. *7*∗*size t*) + *length ts*
      by(*simp add*: *sum_list_filter_le_nat*)
    also have ... = ($\sum$ *t*←*ts*. *7* ∗ *size t* + *1*)
      by(*simp add*: *sum_list_Suc*)
    finally show *?case* **.**
  qed
qed

end

39 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*

definition *empty* :: *'a trie* **where**
[*simp*]: *empty* = *Nd False* (λ_. *None*)

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 list* ⇒ *'a trie* ⇒ *'a trie* **where**
insert [] (*Nd b m*) = *Nd True m* |
insert (*x*#*xs*) (*Nd b m*) =
  (*let s* = (*case m x of None* ⇒ *empty* | *Some t* ⇒ *t*) *in Nd b* (*m*(*x* := *Some*(*insert xs s*))))

fun *delete* :: *'a 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*)))

Use (a tuned version of) *isin* as an abstraction function:

```
lemma isin_case: isin (Nd b m) xs =
  (case xs of
    [] ⇒ b |
    x # ys ⇒ (case m x of None ⇒ False | Some t ⇒ isin t ys))
by(cases xs)auto

definition set :: 'a trie ⇒ 'a list set where
[simp]: set t = {xs. isin t xs}

lemma isin_set: isin t xs = (xs ∈ set t)
by simp

lemma set_insert: set (insert xs t) = set t ∪ {xs}
by (induction xs t rule: insert.induct)
   (auto simp: isin_case split!: if_splits option.splits list.splits)

lemma set_delete: set (delete xs t) = set t − {xs}
by (induction xs t rule: delete.induct)
   (auto simp: isin_case split!: if_splits option.splits list.splits)

interpretation S: Set
where empty = empty 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 add: isin_case split: list.split)
next
  case 2 show ?case by(rule isin_set)
next
  case 3 show ?case by(rule set_insert)
next
  case 4 show ?case by(rule set_delete)
qed (rule TrueI)+

end
```

40 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*.

term *size_tree*
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*(*Suc* (*size* (*snd*(*fst ab*))))) *t*)
apply(*induction t a rule*: *lookup.induct*)
apply(*auto split*: *if_splits*)
done

definition *empty* :: *'a trie_map* **where**
[*simp*]: *empty = Nd False Leaf*

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* ⇒ *empty* | *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*)

40.1 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* (*λa*. *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* = *empty* **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 add*: *isin_case split*: *list.split*)
next
  case *2* **thus** *?case* **by** (*simp add*: *isin_abs*)
next
  case *3* **thus** *?case* **by** (*simp add*: *set_insert abs_insert del*: *set_def*)
next
  case *4* **thus** *?case* **by** (*simp add*: *set_delete abs_delete del*: *set_def*)
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

41 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*))

41.1 Trie

datatype *trie* = *Lf* | *Nd bool trie* ∗ *trie*

definition *empty* :: *trie* **where**
[*simp*]: *empty* = *Lf*

```
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 xs t) ys = (xs = ys ∨ isin t ys)
apply(induction xs t arbitrary: ys 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 xs t) ys = (xs ≠ ys ∧ isin t ys)
```

```
apply(induction xs t arbitrary: ys 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_empty*: *set_trie empty* = {}
by(*simp add*: *set_trie_def*)

lemma *set_trie_isin*: *isin t xs* = (*xs* ∈ *set_trie t*)
by(*simp add*: *set_trie_def*)

lemma *set_trie_insert*: *set_trie*(*insert xs t*) = *set_trie t* ∪ {*xs*}
by(*auto simp add*: *isin_insert set_trie_def*)

lemma *set_trie_delete*: *set_trie*(*delete xs t*) = *set_trie t* − {*xs*}
by(*auto simp add*: *isin_delete set_trie_def*)

```
interpretation S: Set
where empty = empty and isin = isin and insert = insert and delete =
delete
and set = set_trie and invar = λt. True
proof (standard, goal_cases)
  case 1 show ?case by (rule set_trie_empty)
next
  case 2 show ?case by(rule set_trie_isin)
next
  case 3 thus ?case by(auto simp: set_trie_insert)
next
  case 4 show ?case by(rule set_trie_delete)
qed (rule TrueI)+
```

41.2 Patricia Trie

datatype *trieP* = *LfP* | *NdP bool list bool trieP* ∗ *trieP*

```
fun isinP :: trieP ⇒ 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)
```

definition *emptyP* :: *trieP* **where**
[*simp*]: *emptyP* = *LfP*

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'*; $\bigwedge$*a b*. *lr'*=(*a*,*b*) ⟹ *f* (*a*) = *f'* (*a*) ; $\bigwedge$*a b*. *lr'*=(*a*,*b*) ⟹ *f* (*b*) = *f'* (*b*) ⟧
 ⟹ *mod2 f k lr*= *mod2 f' k' lr'*
by(*cases lr*, *cases lr'*, *auto*)

fun *insertP* :: *bool list* ⇒ *trieP* ⇒ *trieP* **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* ⇒ *trieP* ∗ *trieP* ⇒ *trieP* **where**
nodeP ps b lr = (*if* ¬ *b* ∧ *lr* = (*LfP*,*LfP*) *then LfP else NdP ps b lr*)

fun *deleteP* :: *bool list* ⇒ *trieP* ⇒ *trieP* **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)
```

41.2.1 Functional Correctness

First step: *trieP* implements *trie* via the abstraction function *abs_trieP*:

```
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_trieP :: trieP ⇒ trie where
abs_trieP LfP = Lf |
abs_trieP (NdP ps b (l,r)) = prefix_trie ps (Nd b (abs_trieP l, abs_trieP
r))
```

Correctness of *isinP*:

```
lemma isin_prefix_trie:
 isin (prefix_trie ps t) ks
  = (ps = take (length ps) ks ∧ isin t (drop (length ps) ks))
apply(induction ps arbitrary: ks)
apply(auto split: list.split)
done

lemma abs_trieP_isinP:
 isinP t ks = isin (abs_trieP t) ks
apply(induction t arbitrary: ks rule: abs_trieP.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'*) $\Longrightarrow$
  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_trieP_insertP*:
  abs_trieP (*insertP ks t*) = *insert ks* (*abs_trieP 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* $\longleftrightarrow$ *xs* = [] $\wedge$ *t* = *Lf*
by(*cases xs*)(*auto*)

lemma *abs_trieP_Lf*: *abs_trieP t* = *Lf* $\longleftrightarrow$ *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_trieP*:

  delete ks (*abs_trieP t*) = *abs_trieP* (*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_trieP_Lf
    dest!: *split_if split*: *if_splits list.split prod.split*)
done

The overall correctness proof. Simply composes correctness lemmas.

definition *set_trieP* :: *trieP* ⇒ *bool list set* **where**
set_trieP = *set_trie o abs_trieP*

lemma *isinP_set_trieP*: *isinP t xs* = (*xs* ∈ *set_trieP t*)
by(*simp add*: *abs_trieP_isinP set_trie_isin set_trieP_def*)

lemma *set_trieP_insertP*: *set_trieP* (*insertP xs t*) = *set_trieP t* ∪ {*xs*}
by(*simp add*: *abs_trieP_insertP set_trie_insert set_trieP_def*)

lemma *set_trieP_deleteP*: *set_trieP* (*deleteP xs t*) = *set_trieP t* − {*xs*}
by(*auto simp*: *set_trie_delete set_trieP_def simp flip*: *delete_abs_trieP*)

interpretation *SP*: *Set*
where *empty* = *emptyP* **and** *isin* = *isinP* **and** *insert* = *insertP* **and** *delete* = *deleteP*
and *set* = *set_trieP* **and** *invar* = λ*t*. *True*
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** (*simp add*: *set_trieP_def set_trie_def*)
next
  case *2* **show** *?case* **by**(*rule isinP_set_trieP*)
next
  case *3* **thus** *?case* **by** (*auto simp*: *set_trieP_insertP*)
next
  case *4* **thus** *?case* **by**(*auto simp*: *set_trieP_deleteP*)
qed (*rule TrueI*)+

end

42 Queue Specification

theory *Queue_Spec*
imports *Main*
begin

The basic queue interface with *list*-based specification:

locale *Queue* =

fixes *empty* :: $'q$
fixes *enq* :: $'a \Rightarrow 'q \Rightarrow 'q$
fixes *first* :: $'q \Rightarrow 'a$
fixes *deq* :: $'q \Rightarrow 'q$
fixes *is_empty* :: $'q \Rightarrow$ *bool*
fixes *list* :: $'q \Rightarrow 'a$ *list*
fixes *invar* :: $'q \Rightarrow$ *bool*
assumes *list_empty*: *list empty* = []
assumes *list_enq*: *invar q* $\Longrightarrow$ *list*(*enq x q*) = *list q* @ [*x*]
assumes *list_deq*: *invar q* $\Longrightarrow$ *list*(*deq q*) = *tl*(*list q*)
assumes *list_first*: *invar q* $\Longrightarrow \neg$ *list q* = [] $\Longrightarrow$ *first q* = *hd*(*list q*)
assumes *list_is_empty*: *invar q* $\Longrightarrow$ *is_empty q* = (*list q* = [])
assumes *invar_empty*: *invar empty*
assumes *invar_enq*: *invar q* $\Longrightarrow$ *invar*(*enq x q*)
assumes *invar_deq*: *invar q* $\Longrightarrow$ *invar*(*deq q*)

end
theory *Reverse*
imports *Main*
begin

fun *T_append* :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *nat* **where**
T_append [] *ys* = *1* |
T_append (*x#xs*) *ys* = *T_append xs ys* + *1*

fun *T_rev* :: $'a$ *list* $\Rightarrow$ *nat* **where**
T_rev [] = *1* |
T_rev (*x#xs*) = *T_rev xs* + *T_append* (*rev xs*) [*x*] + *1*

lemma *T_append*: *T_append xs ys* = *length xs* + *1*
by(*induction xs*) *auto*

lemma *T_rev*: *T_rev xs* $\leq$ (*length xs* + *1*)^*2*
by(*induction xs*) (*auto simp*: *T_append power2_eq_square*)

fun *itrev* :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* **where**
itrev [] *ys* = *ys* |
itrev (*x#xs*) *ys* = *itrev xs* (*x* # *ys*)

lemma *itrev*: *itrev xs ys* = *rev xs* @ *ys*
by(*induction xs arbitrary*: *ys*) *auto*

lemma *itrev_Nil*: *itrev xs* [] = *rev xs*
by(*simp add*: *itrev*)

fun *T_itrev* :: *'a list* ⇒ *'a list* ⇒ *nat* **where**
T_itrev [] *ys* = *1* |
T_itrev (*x#xs*) *ys* = *T_itrev xs* (*x* # *ys*) + *1*

lemma *T_itrev*: *T_itrev xs ys* = *length xs* + *1*
by(*induction xs arbitrary*: *ys*) *auto*

end

43 Queue Implementation via 2 Lists

theory *Queue_2Lists*
imports
  Queue_Spec
  Reverse
begin

Definitions:

type_synonym *'a queue* = *'a list* × *'a list*

fun *norm* :: *'a queue* ⇒ *'a queue* **where**
norm (*fs*,*rs*) = (*if fs* = [] *then* (*itrev rs* [], []) *else* (*fs*,*rs*))

fun *enq* :: *'a* ⇒ *'a queue* ⇒ *'a queue* **where**
enq a (*fs*,*rs*) = *norm*(*fs*, *a* # *rs*)

fun *deq* :: *'a queue* ⇒ *'a queue* **where**
deq (*fs*,*rs*) = (*if fs* = [] *then* (*fs*,*rs*) *else norm*(*tl fs*,*rs*))

fun *first* :: *'a queue* ⇒ *'a* **where**
first (*a* # *fs*,*rs*) = *a*

fun *is_empty* :: *'a queue* ⇒ *bool* **where**
is_empty (*fs*,*rs*) = (*fs* = [])

fun *list* :: *'a queue* ⇒ *'a list* **where**
list (*fs*,*rs*) = *fs* @ *rev rs*

fun *invar* :: *'a queue* ⇒ *bool* **where**
invar (*fs*,*rs*) = (*fs* = [] ⟶ *rs* = [])

Implementation correctness:

interpretation *Queue*

```
where empty = ([],[]) and enq = enq and deq = deq and first = first
and is_empty = is_empty and list = list and invar = invar
proof (standard, goal_cases)
  case 1 show ?case by (simp)
next
  case (2 q) thus ?case by(cases q) (simp)
next
  case (3 q) thus ?case by(cases q) (simp add: itrev_Nil)
next
  case (4 q) thus ?case by(cases q) (auto simp: neq_Nil_conv)
next
  case (5 q) thus ?case by(cases q) (auto)
next
  case 6 show ?case by(simp)
next
  case (7 q) thus ?case by(cases q) (simp)
next
  case (8 q) thus ?case by(cases q) (simp)
qed
```

Running times:

```
fun T_norm :: 'a queue ⇒ nat where
T_norm (fs,rs) = (if fs = [] then T_itrev rs [] else 0) + 1

fun T_enq :: 'a ⇒ 'a queue ⇒ nat where
T_enq a (fs,rs) = T_norm(fs, a # rs) + 1

fun T_deq :: 'a queue ⇒ nat where
T_deq (fs,rs) = (if fs = [] then 0 else T_norm(tl fs,rs)) + 1

fun T_first :: 'a queue ⇒ nat where
T_first (a # fs,rs) = 1

fun T_is_empty :: 'a queue ⇒ nat where
T_is_empty (fs,rs) = 1
```

Amortized running times:

```
fun Φ :: 'a queue ⇒ nat where
Φ(fs,rs) = length rs

lemma a_enq: T_enq a (fs,rs) + Φ(enq a (fs,rs)) − Φ(fs,rs) ≤ 4
by(auto simp: T_itrev)

lemma a_deq: T_deq (fs,rs) + Φ(deq (fs,rs)) − Φ(fs,rs) ≤ 3
```

by(*auto simp*: *T_itrev*)

end

44 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

45 Heaps

theory *Heaps*

```
imports
  HOL−Library.Tree_Multiset
  Priority_Queue_Specs
begin
```

Heap = priority queue on trees:

```
locale Heap =
fixes insert :: ('a::linorder) ⇒ 'a tree ⇒ 'a tree
and  del_min :: 'a tree ⇒ 'a tree
assumes mset_insert: heap q ⟹ mset_tree (insert x q) = {#x#} +
mset_tree q
and    mset_del_min: ⟦ heap q;  q ≠ Leaf ⟧ ⟹ mset_tree (del_min q) =
mset_tree q − {#value q#}
and     heap_insert: heap q ⟹ heap(insert x q)
and    heap_del_min: heap q ⟹ heap(del_min q)
begin

definition empty :: 'a tree where
empty = Leaf

fun is_empty :: 'a tree ⇒ bool where
is_empty t = (t = Leaf)

sublocale Priority_Queue where empty = empty and is_empty = is_empty
and insert = insert
and get_min = value and del_min = del_min and invar = heap and
mset = mset_tree
proof (standard, goal_cases)
  case 1 thus ?case by (simp add: empty_def)
next
  case 2 thus ?case by(auto)
next
  case 3 thus ?case by(simp add: mset_insert)
next
  case 4 thus ?case by(simp add: mset_del_min)
next
   case 5 thus ?case by(auto simp: neq_Leaf_iff Min_insert2 simp del:
Un_iff)
next
  case 6 thus ?case by(simp add: empty_def)
next
  case 7 thus ?case by(simp add: heap_insert)
next
  case 8 thus ?case by(simp add: heap_del_min)
```

qed

end

Once you have *merge*, *insert* and *del_min* are easy:

locale *Heap_Merge* =
fixes *merge* :: *'a::linorder tree* ⇒ *'a tree* ⇒ *'a tree*
assumes *mset_merge*: ⟦ *heap q1*; *heap q2* ⟧ ⟹ *mset_tree* (*merge q1 q2*) = *mset_tree q1* + *mset_tree q2*
and *invar_merge*: ⟦ *heap q1*; *heap q2* ⟧ ⟹ *heap* (*merge q1 q2*)
begin

fun *insert* :: *'a* ⇒ *'a tree* ⇒ *'a tree* **where**
insert x t = *merge* (*Node Leaf x Leaf*) *t*

fun *del_min* :: *'a tree* ⇒ *'a tree* **where**
del_min Leaf = *Leaf* |
del_min (*Node l x r*) = *merge l r*

interpretation *Heap insert del_min*
proof(*standard*, *goal_cases*)
  case *1* **thus** *?case* **by**(*simp add*:*mset_merge*)
next
  case (*2 q*) **thus** *?case* **by**(*cases q*)(*auto simp*: *mset_merge*)
next
  case *3* **thus** *?case* **by** (*simp add*: *invar_merge*)
next
  case (*4 q*) **thus** *?case* **by** (*cases q*)(*auto simp*: *invar_merge*)
qed

sublocale *PQM*: *Priority_Queue_Merge* **where** *empty* = *empty* **and** *is_empty* = *is_empty* **and** *insert* = *insert*
and *get_min* = *value* **and** *del_min* = *del_min* **and** *invar* = *heap* **and** *mset* = *mset_tree* **and** *merge* = *merge*
proof(*standard*, *goal_cases*)
  case *1* **thus** *?case* **by** (*simp add*: *mset_merge*)
next
  case *2* **thus** *?case* **by** (*simp add*: *invar_merge*)
qed

end

end

46 Leftist Heap

theory *Leftist_Heap*
imports
  HOL−Library.Pattern_Aliases
  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 *mht* :: *'a lheap* ⇒ *nat* **where**
mht Leaf = *0* |
mht (*Node* _ (_, *n*) _) = *n*

The invariants:

fun (**in** *linorder*) *heap* :: (*'a*∗*'b*) *tree* ⇒ *bool* **where**
heap Leaf = *True* |
heap (*Node l* (*m*, _) *r*) =
  ((∀ *x* ∈ *set_tree l* ∪ *set_tree r*. *m* ≤ *x*) ∧ *heap l* ∧ *heap r*)

fun *ltree* :: *'a lheap* ⇒ *bool* **where**
ltree Leaf = *True* |
ltree (*Node l* (*a*, *n*) *r*) =
 (*min_height l* ≥ *min_height r* ∧ *n* = *min_height r* + *1* ∧ *ltree l* & *ltree r*)

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

definition *node* :: *'a lheap* ⇒ *'a* ⇒ *'a lheap* ⇒ *'a lheap* **where**
node l a r =
 (*let mhl* = *mht l*; *mhr* = *mht r*
  in if mhl ≥ *mhr then Node l* (*a*,*mhr*+*1*) *r else Node r* (*a*,*mhl*+*1*) *l*)

fun *get_min* :: *'a lheap* ⇒ *'a* **where**
get_min(*Node l* (*a*, *n*) *r*) = *a*

For function *merge*:

unbundle *pattern_aliases*

fun *merge* :: *'a::ord lheap* ⇒ *'a lheap* ⇒ *'a lheap* **where**
merge Leaf t = *t* |
merge t Leaf = *t* |
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*))

Termination of *merge*: by sum or lexicographic product of the sizes of the two arguments. Isabelle uses a lexicographic product.

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* _ *r*) = *merge l r*

46.1 Lemmas

lemma *mset_tree_empty*: *mset_tree t* = {#} ⟷ *t* = *Leaf*
by(*cases t*) *auto*

lemma *mht_eq_min_height*: *ltree t* ⟹ *mht t* = *min_height t*
by(*cases t*) *auto*

lemma *ltree_node*: *ltree* (*node l a r*) ⟷ *ltree l* ∧ *ltree r*
by(*auto simp add*: *node_def mht_eq_min_height*)

lemma *heap_node*: *heap* (*node l a r*) ⟷
 heap l ∧ *heap r* ∧ (∀ *x* ∈ *set_tree l* ∪ *set_tree r*. *a* ≤ *x*)
by(*auto simp add*: *node_def*)

lemma *set_tree_mset*: *set_tree t* = *set_mset*(*mset_tree t*)
by(*induction t*) *auto*

46.2 Functional Correctness

lemma *mset_merge*: *mset_tree* (*merge t1 t2*) = *mset_tree t1* + *mset_tree t2*
by (*induction t1 t2 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 t*; *t* ≠ *Leaf* ⟧ ⟹ *get_min t* = *Min*(*set_tree t*)
by (*cases t*) (*auto simp add*: *eq_Min_iff*)

lemma *mset_del_min*: *mset_tree* (*del_min t*) = *mset_tree t* − {# *get_min t* #}
by (*cases t*) (*auto simp*: *mset_merge*)

lemma *ltree_merge*: ⟦ *ltree l*; *ltree r* ⟧ ⟹ *ltree* (*merge l r*)
by(*induction l r rule*: *merge.induct*)(*auto simp*: *ltree_node*)

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 set_tree_mset*)
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* = *empty* **and** *is_empty* = *λt. t* = *Leaf*
and *insert* = *insert* **and** *del_min* = *del_min*
and *get_min* = *get_min* **and** *merge* = *merge*
and *invar* = *λt. heap t* ∧ *ltree t* **and** *mset* = *mset_tree*

proof(*standard*, *goal_cases*)
  case *1* **show** *?case* **by** (*simp add*: *empty_def*)
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 set_tree_mset*)
next
  case *6* **thus** *?case* **by**(*simp add*: *empty_def*)
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

46.3 Complexity

Explicit termination argument: sum of sizes

fun *T_merge* :: *'a::ord lheap* ⇒ *'a lheap* ⇒ *nat* **where**
T_merge Leaf t = *1* |
T_merge t Leaf = *1* |
T_merge (*Node l1* (*a1*, *n1*) *r1* =: *t1*) (*Node l2* (*a2*, *n2*) *r2* =: *t2*) =
  (*if a1* ≤ *a2 then T_merge r1 t2*
   else T_merge t1 r2) + *1*

definition *T_insert* :: *'a::ord* ⇒ *'a lheap* ⇒ *nat* **where**
T_insert x t = *T_merge* (*Node Leaf* (*x*, *1*) *Leaf*) *t* + *1*

fun *T_del_min* :: *'a::ord lheap* ⇒ *nat* **where**
T_del_min Leaf = *1* |
T_del_min (*Node l _ r*) = *T_merge l r* + *1*

lemma *T_merge_min_height*: *ltree l* ⟹ *ltree r* ⟹ *T_merge l r* ≤ *min_height l* + *min_height r* + *1*
proof(*induction l r rule*: *merge.induct*)
  case *3* **thus** *?case* **by**(*auto*)
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 min_height_size1[of l]]
  le_log2_of_power[OF min_height_size1[of r]] T_merge_min_height[of
l r] assms
by linarith

corollary T_insert_log: ltree t ⟹ T_insert x t ≤ log 2 (size1 t) + 3
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 rule: tree2_cases)
  case Leaf thus ?thesis using assms by simp
next
  case [simp]: (Node l _ _ r)
  have T_del_min t = T_merge l r + 1 by simp
  also have ... ≤ log 2 (size1 l) + log 2 (size1 r) + 2
    using ‹ltree t› T_merge_log[of l r] by (auto simp del: T_merge.simps)
  also have ... ≤ 2 ∗ log 2 (size1 t) + 1
    using ld_ld_1_less[of size1 l size1 r] by (simp)
  finally show ?thesis .
qed

end
```

47 Binomial Heap

theory *Binomial_Heap*
imports
  HOL−Library.Pattern_Aliases
  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.

47.1 Binomial Tree and Heap Datatype

datatype *'a tree = Node (rank: nat) (root: 'a) (children: 'a tree list)*

type_synonym *'a heap = 'a tree list*

47.1.1 Multiset of elements

fun *mset_tree* :: *'a::linorder tree ⇒ 'a multiset* **where**
  mset_tree (Node _ a ts) = {#a#} + (∑ t∈#mset ts. mset_tree t)

definition *mset_heap* :: *'a::linorder heap ⇒ 'a multiset* **where**
  mset_heap ts = (∑ t∈#mset ts. mset_tree t)

lemma *mset_tree_simp_alt[simp]*:
  mset_tree (Node r a ts) = {#a#} + mset_heap ts
  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*)

47.1.2 Invariants

Binomial tree

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*]

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_tree t* ⟷ *invar_btree t* ∧ *invar_otree t*

Binomial Heap invariant

definition *invar ts* ⟷ (∀ *t*∈*set ts*. *invar_tree t*) ∧ (*sorted_wrt* (<) (*map rank ts*))

The children of a node are a valid heap

lemma *invar_children*:
  invar_tree (*Node r v ts*) ⟹ *invar* (*rev ts*)
  by (*auto simp*: *invar_tree_def invar_def rev_map*[*symmetric*])

47.2 Operations and Their Functional Correctness

47.2.1 *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_link*:

  assumes *invar_tree* t_1
  assumes *invar_tree* t_2
  assumes *rank* t_1 = *rank* t_2
  shows *invar_tree* (*link* t_1 t_2)
using *assms* **unfolding** *invar_tree_def*
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*

47.2.2 *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_tree0*[*simp*]: *invar_tree* (*Node 0 x* [])
unfolding *invar_tree_def* **by** *auto*

lemma *invar_Cons*[*simp*]:
  invar (*t*#*ts*)
  ⟷ *invar_tree t* ∧ *invar ts* ∧ (∀ *t'*∈*set ts*. *rank t* < *rank t'*)
by (*auto simp*: *invar_def*)

lemma *invar_ins_tree*:
  assumes *invar_tree t*
  assumes *invar ts*
  assumes ∀ *t'*∈*set ts*. *rank t* ≤ *rank t'*
  shows *invar* (*ins_tree t ts*)
using *assms*
by (*induction t ts rule*: *ins_tree.induct*) (*auto simp*: *invar_link less_eq_Suc_le*[*symmetric*])

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'* ∈ *set* (*ins_tree t ts*)
  assumes ∀ *t'*∈*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*)

47.2.3 *insert*

hide_const (**open**) *insert*

definition *insert* :: *′a*::*linorder* ⇒ *′a heap* ⇒ *′a heap* **where**
insert x ts = *ins_tree* (*Node 0 x* []) *ts*

lemma *invar_insert*[*simp*]: *invar t* ⟹ *invar* (*insert x t*)
by (*auto intro*!: *invar_ins_tree simp*: *insert_def*)

lemma *mset_heap_insert*[*simp*]: *mset_heap* (*insert x t*) = {#*x*#} + *mset_heap*
t
by(*auto simp*: *insert_def*)

47.2.4 *merge*

context
includes *pattern_aliases*
begin

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 =: h_1) (t_2#ts_2 =: h_2) = (
    if rank t_1 < *rank* t_2 *then* t_1 # *merge* ts_1 h_2 *else*
    if rank t_2 < *rank* t_1 *then* t_2 # *merge* h_1 ts_2
    else ins_tree (*link* t_1 t_2) (*merge* ts_1 ts_2)
)

end

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_1∈*set* ts_1. *rank t* < *rank* t_1
  assumes ∀ t_2∈*set* ts_2. *rank t* < *rank* t_2
  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_merge*[*simp*]:
 assumes *invar* ts_1
 assumes *invar* ts_2
 shows *invar* (*merge* ts_1 ts_2)
using *assms*
by (*induction* ts_1 ts_2 *rule*: *merge.induct*)
 (*auto 0 3 simp*: *Suc_le_eq intro*!: *invar_ins_tree invar_link elim*!: *merge_rank_bound*)

Longer, more explicit proof of *invar_merge*, to illustrate the application of the *merge_rank_bound* lemma.

lemma
 assumes *invar* ts_1
 assumes *invar* ts_2
 shows *invar* (*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)
 — Invariants of the parts can be shown automatically
 from *3.prems* **have** [*simp*]:
  invar_tree t_1 *invar_tree* t_2

  by *auto*

 — These are the three cases of the *merge* function
 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*
  — *merge* takes the first tree from the left heap
  then have *merge* (t_1 # ts_1) (t_2 # ts_2) = t_1 # *merge* ts_1 (t_2 # ts_2) **by** *simp*
  also have *invar* ... **proof** (*simp*, *intro conjI*)
   — Invariant follows from induction hypothesis
   show *invar* (*merge* ts_1 (t_2 # ts_2))
    using *LT 3.IH 3.prems* **by** *simp*

   — It remains to show that t_1 has smallest rank.
   show ∀ t'∈*set* (*merge* ts_1 (t_2 # ts_2)). *rank* $t_1 <$ *rank* t'

— Which is done by auxiliary lemma *merge_rank_bound*
using *LT 3.prems* **by** (*force elim*!: *merge_rank_bound*)
qed
finally show *?thesis* **.**
next
— *merge* takes the first tree from the right heap
case *GT*
— The proof is anaologous to the *LT* case
then show *?thesis* **using** *3.prems 3.IH* **by** (*force elim*!: *merge_rank_bound*)
next
case [*simp*]: *EQ*
— *merge* links both first trees, and inserts them into the merged remaining heaps
have *merge* $(t_1 \# ts_1)$ $(t_2 \# ts_2)$ = *ins_tree* (*link* t_1 t_2) (*merge* ts_1 ts_2) **by** *simp*
also have *invar* . . . **proof** (*intro invar_ins_tree invar_link*)
— Invariant of merged remaining heaps follows by IH
show *invar* (*merge* ts_1 ts_2)
using *EQ 3.prems 3.IH* **by** *auto*

— For insertion, we have to show that the rank of the linked tree is ≤ the ranks in the merged remaining heaps
show $\forall t' \in$ *set* (*merge* ts_1 ts_2). *rank* (*link* t_1 t_2) ≤ *rank* t'
proof −
— Which is, again, done with the help of *merge_rank_bound*
have *rank* (*link* t_1 t_2) = *Suc* (*rank* t_2) **by** *simp*
thus *?thesis* **using** *3.prems* **by** (*auto simp*: *Suc_le_eq elim*!: *merge_rank_bound*)
qed
qed *simp_all*
finally show *?thesis* **.**
qed
qed *auto*

lemma *mset_heap_merge*[*simp*]:
mset_heap (*merge* ts_1 ts_2) = *mset_heap* ts_1 + *mset_heap* ts_2
by (*induction* ts_1 ts_2 *rule*: *merge.induct*) *auto*

47.2.5 *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_tree_root_min:
  assumes invar_tree t
  assumes x ∈# mset_tree t
  shows root t ≤ x
using assms unfolding invar_tree_def
by (induction t arbitrary: x rule: mset_tree.induct) (fastforce simp: mset_heap_def)

lemma get_min_mset:
  assumes ts≠[]
  assumes invar 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_tree_root_min min_def intro: order_trans;
      meson linear order_trans invar_tree_root_min
      )+
done

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)
```

47.2.6 *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_get_min_rest*:
  assumes *get_min_rest ts* = (*t′*,*ts′*)
  assumes *ts*≠[]
  assumes *invar ts*
  shows *invar_tree t′* **and** *invar ts′*
proof −
  have *invar_tree t′* ∧ *invar 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_tree t′* **and** *invar ts′* **by** *auto*
qed

47.2.7 *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 del_min_def
by (auto
      split: prod.split tree.split
      intro!: invar_merge invar_children
      dest: invar_get_min_rest
    )

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
```

47.2.8 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)
```

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

47.3 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

lemma size_mset_tree:
  assumes invar_tree t
  shows size (mset_tree t) = 2^rank t
using assms unfolding invar_tree_def
by (simp add: size_mset_btree)
```

The length of a binomial heap is bounded by the number of its elements

```
lemma size_mset_heap:
  assumes invar ts
  shows length ts ≤ log 2 (size (mset_heap ts) + 1)
proof –
  from ‹invar ts› have
    ASC: sorted_wrt (<) (map rank ts) and
    TINV: ∀ t∈set ts. invar_tree t
    unfolding invar_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_tree)
  also have ... = size (mset_heap ts) + 1
    unfolding mset_heap_def by (induction ts) auto
  finally have 2^length ts ≤ size (mset_heap ts) + 1 .
  then show ?thesis using le_log2_of_power by blast
qed
```

47.3.1 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
```

This function is non-canonical: we omitted a *+1* in the *else*-part, to keep the following analysis simpler and more to the point.

```
fun T_ins_tree :: 'a::linorder tree ⇒ 'a heap ⇒ nat where
  T_ins_tree t [] = 1
| T_ins_tree t₁ (t₂ # ts) = (
    (if rank t₁ < rank t₂ then 1
     else T_link t₁ t₂ + T_ins_tree (link t₁ t₂) ts)
  )

definition T_insert :: 'a::linorder ⇒ 'a heap ⇒ nat where
T_insert x ts = T_ins_tree (Node 0 x []) ts + 1
```

lemma *T_ins_tree_simple_bound*: *T_ins_tree t ts* $\leq$ *length ts* + *1*
by (*induction t ts rule*: *T_ins_tree.induct*) *auto*

47.3.2 *T_insert*

lemma *T_insert_bound*:
  assumes *invar ts*
  shows *T_insert x ts* $\leq$ *log 2* (*size* (*mset_heap ts*) + *1*) + *2*
proof –
  have *real* (*T_insert x ts*) $\leq$ *real* (*length ts*) + *2*
    unfolding *T_insert_def* **using** *T_ins_tree_simple_bound*
    using *of_nat_mono* **by** *fastforce*
  also note *size_mset_heap*[*OF* ‹*invar ts*›]
  finally show *?thesis* **by** *simp*
qed

47.3.3 *T_merge*

context
includes *pattern_aliases*
begin

fun *T_merge* :: *'a::linorder heap* ⇒ *'a heap* ⇒ *nat* **where**
  T_merge ts_1 [] = *1*
| *T_merge* [] ts_2 = *1*
| *T_merge* (t_1#ts_1 =: h_1) (t_2#ts_2 =: h_2) = *1* + (
    if rank t_1 < *rank* t_2 *then T_merge* ts_1 h_2
    else if rank t_2 < *rank* t_1 *then T_merge* h_1 ts_2
    else T_ins_tree (*link* t_1 t_2) (*merge* ts_1 ts_2) + *T_merge* ts_1 ts_2
)

end

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:
  fixes ts1 ts2
  defines n1 ≡ size (mset_heap ts1)
  defines n2 ≡ size (mset_heap ts2)
  assumes invar ts1 invar ts2
  shows T_merge ts1 ts2 ≤ 4*log 2 (n1 + n2 + 1) + 1
proof -
  note n_defs = assms(1,2)

  have T_merge ts1 ts2 ≤ 2 * real (length ts1) + 2 * real (length ts2) + 1
    using T_merge_length[of ts1 ts2] by simp
  also note size_mset_heap[OF ‹invar ts1›]
  also note size_mset_heap[OF ‹invar ts2›]
  finally have T_merge ts1 ts2 ≤ 2 * log 2 (n1 + 1) + 2 * log 2 (n2 +
1) + 1
    unfolding n_defs by (simp add: algebra_simps)
  also have log 2 (n1 + 1) ≤ log 2 (n1 + n2 + 1)
    unfolding n_defs by (simp add: algebra_simps)
  also have log 2 (n2 + 1) ≤ log 2 (n1 + n2 + 1)
    unfolding n_defs by (simp add: algebra_simps)
  finally show ?thesis by (simp add: algebra_simps)
qed
```

47.3.4 *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 note size_mset_heap[OF ‹invar ts›]
  finally show ?thesis .
qed
```

47.3.5 *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*:
  assumes *invar 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 note *size_mset_heap*[*OF* ‹*invar ts*›]
  finally show *?thesis* **.**
qed

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
) + *1*

lemma *T_del_min_bound*:
  fixes *ts*
  defines *n* ≡ *size* (*mset_heap ts*)
  assumes *invar ts* **and** *ts*≠[]
  shows *T_del_min ts* ≤ *6* ∗ *log 2* (*n*+*1*) + *3*
proof −
  obtain *r x* ts_1 ts_2 **where** *GM*: *get_min_rest ts* = (*Node r x* ts_1, ts_2)
    by (*metis surj_pair tree.exhaust_sel*)

  have *I1*: *invar* (*rev* ts_1) **and** *I2*: *invar* ts_2
    using *invar_get_min_rest*[*OF GM* ‹*ts*≠[]› ‹*invar ts*›] *invar_children*
    by *auto*

define n_1 **where** n_1 = *size* (*mset_heap* ts_1)
define n_2 **where** n_2 = *size* (*mset_heap* ts_2)

have $n_1 \leq n$ $n_1 + n_2 \leq n$ **unfolding** *n_def* n_1*_def* n_2*_def*
using *mset_get_min_rest*[*OF GM* ‹*ts*≠[]›]
by (*auto simp*: *mset_heap_def*)

have *T_del_min ts* = *real* (*T_get_min_rest ts*) + *real* (*T_rev* ts_1) + *real* (*T_merge* (*rev* ts_1) ts_2) + *1*
unfolding *T_del_min_def GM*
by *simp*
also have *T_get_min_rest ts* ≤ *log 2* (*n+1*)
using *T_get_min_rest_bound*[*OF* ‹*invar ts*› ‹*ts*≠[]›] **unfolding** *n_def* **by** *simp*
also have *T_rev* ts_1 ≤ *1* + *log 2* (n_1 + *1*)
unfolding *T_rev_def* n_1*_def* **using** *size_mset_heap*[*OF I1*] **by** *simp*
also have *T_merge* (*rev* ts_1) ts_2 ≤ *4∗log 2* (n_1 + n_2 + *1*) + *1*
unfolding n_1*_def* n_2*_def* **using** *T_merge_bound*[*OF I1 I2*] **by** (*simp add*: *algebra_simps*)
finally have *T_del_min ts* ≤ *log 2* (*n+1*) + *log 2* (n_1 + *1*) + *4∗log 2* (*real* (n_1 + n_2) + *1*) + *3*
by (*simp add*: *algebra_simps*)
also note ‹$n_1 + n_2 \leq n$›
also note ‹$n_1 \leq n$›
finally show *?thesis* **by** (*simp add*: *algebra_simps*)
qed

end

48 Time functions for various standard library operations

theory *Time_Funs*
imports *Main*
begin

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

lemma *T_length_eq*: *T_length xs* = *length xs* + *1*
by (*induction xs*) *auto*

lemmas [*simp del*] = *T_length.simps*

fun *T_map* :: (*'a* ⇒ *nat*) ⇒ *'a list* ⇒ *nat* **where**
  T_map T_f [] = *1*
| *T_map T_f* (*x* # *xs*) = *T_f x* + *T_map T_f xs* + *1*

lemma *T_map_eq*: *T_map T_f xs* = ($\sum$ *x*←*xs*. *T_f x*) + *length xs* + *1*
  by (*induction xs*) *auto*

lemmas [*simp del*] = *T_map.simps*

fun *T_filter* :: (*'a* ⇒ *nat*) ⇒ *'a list* ⇒ *nat* **where**
  T_filter T_p [] = *1*
| *T_filter T_p* (*x* # *xs*) = *T_p x* + *T_filter T_p xs* + *1*

lemma *T_filter_eq*: *T_filter T_p xs* = ($\sum$ *x*←*xs*. *T_p x*) + *length xs* +
1
  by (*induction xs*) *auto*

lemmas [*simp del*] = *T_filter.simps*

fun *T_nth* :: *'a list* ⇒ *nat* ⇒ *nat* **where**
  T_nth [] *n* = *1*
| *T_nth* (*x* # *xs*) *n* = (*case n of 0* ⇒ *1* | *Suc n'* ⇒ *T_nth xs n'* + *1*)

lemma *T_nth_eq*: *T_nth xs n* = *min n* (*length xs*) + *1*
  by (*induction xs n rule*: *T_nth.induct*) (*auto split*: *nat.splits*)

lemmas [*simp del*] = *T_nth.simps*

fun *T_take* :: *nat* ⇒ *'a list* ⇒ *nat* **where**
  T_take n [] = *1*
| *T_take n* (*x* # *xs*) = (*case n of 0* ⇒ *1* | *Suc n'* ⇒ *T_take n' xs* + *1*)

lemma *T_take_eq*: *T_take n xs* = *min n* (*length xs*) + *1*
  by (*induction xs arbitrary*: *n*) (*auto split*: *nat.splits*)

fun *T_drop* :: *nat* ⇒ *'a list* ⇒ *nat* **where**
  T_drop n [] = *1*

| *T_drop n* (*x* # *xs*) = (*case n of 0* ⇒ *1* | *Suc n'* ⇒ *T_drop n' xs* + *1*)

lemma *T_drop_eq*: *T_drop n xs* = *min n* (*length xs*) + *1*
  by (*induction xs arbitrary*: *n*) (*auto split*: *nat.splits*)

end

49 The Median-of-Medians Selection Algorithm

theory *Selection*
  imports *Complex_Main Sorting Time_Funs*
begin

Note that there is significant overlap between this theory (which is intended mostly for the Functional Data Structures book) and the Median-of-Medians AFP entry.

49.1 Auxiliary material

lemma *replicate_numeral*: *replicate* (*numeral n*) *x* = *x* # *replicate* (*pred_numeral n*) *x*
  by (*simp add*: *numeral_eq_Suc*)

lemma *isort_correct*: *isort xs* = *sort xs*
  using *sorted_isort mset_isort* **by** (*metis properties_for_sort*)

lemma *sum_list_replicate* [*simp*]: *sum_list* (*replicate n x*) = *n* ∗ *x*
  by (*induction n*) *auto*

lemma *mset_concat*: *mset* (*concat xss*) = *sum_list* (*map mset xss*)
  by (*induction xss*) *simp_all*

lemma *set_mset_sum_list* [*simp*]: *set_mset* (*sum_list xs*) = (⋃*x*∈*set xs*. *set_mset x*)
  by (*induction xs*) *auto*

lemma *filter_mset_image_mset*:
  filter_mset P (*image_mset f A*) = *image_mset f* (*filter_mset* (λ*x*. *P* (*f x*)) *A*)
  by (*induction A*) *auto*

lemma *filter_mset_sum_list*: *filter_mset P* (*sum_list xs*) = *sum_list* (*map* (*filter_mset P*) *xs*)

by $(induction\ xs)\ simp\_all$

lemma $sum\_mset\_mset\_mono$:
assumes $(\bigwedge x.\ x \in\#\ A \Longrightarrow f\ x \subseteq\#\ g\ x)$
shows $(\sum x{\in}\#A.\ f\ x) \subseteq\# (\sum x{\in}\#A.\ g\ x)$
using $assms$ **by** $(induction\ A)\ (auto\ intro!:\ subset\_mset.add\_mono)$

lemma $mset\_filter\_mono$:
assumes $A \subseteq\#\ B\ \bigwedge x.\ x \in\#\ A \Longrightarrow P\ x \Longrightarrow Q\ x$
shows $filter\_mset\ P\ A \subseteq\#\ filter\_mset\ Q\ B$
by $(rule\ mset\_subset\_eqI)\ (insert\ assms,\ auto\ simp:\ mset\_subset\_eq\_count$
$count\_eq\_zero\_iff)$

lemma $size\_mset\_sum\_mset\_distrib$: $size\ (sum\_mset\ A :: 'a\ multiset) =$
$sum\_mset\ (image\_mset\ size\ A)$
by $(induction\ A)\ auto$

lemma sum_mset_mono:
assumes $\bigwedge x.\ x \in\#\ A \Longrightarrow f\ x \le (g\ x :: 'a :: \{ordered\_ab\_semigroup\_add,comm\_monoid\_add\})$
shows $(\sum x{\in}\#A.\ f\ x) \le (\sum x{\in}\#A.\ g\ x)$
using $assms$ **by** $(induction\ A)\ (auto\ intro!:\ add\_mono)$

lemma $filter\_mset\_is\_empty\_iff$: $filter\_mset\ P\ A = \{\#\} \longleftrightarrow (\forall x.\ x \in\#$
$A \longrightarrow \neg P\ x)$
by $(auto\ simp:\ multiset\_eq\_iff\ count\_eq\_zero\_iff)$

lemma $sort\_eq\_Nil\_iff\ [simp]$: $sort\ xs = [] \longleftrightarrow xs = []$
by $(metis\ set\_empty\ set\_sort)$

lemma $sort\_mset\_cong$: $mset\ xs = mset\ ys \Longrightarrow sort\ xs = sort\ ys$
by $(metis\ sorted\_list\_of\_multiset\_mset)$

lemma Min_set_sorted: $sorted\ xs \Longrightarrow xs \neq [] \Longrightarrow Min\ (set\ xs) = hd\ xs$
by $(cases\ xs;\ force\ intro:\ Min\_insert2)$

lemma hd_sort:
fixes $xs :: 'a :: linorder\ list$
shows $xs \neq [] \Longrightarrow hd\ (sort\ xs) = Min\ (set\ xs)$
by $(subst\ Min\_set\_sorted\ [symmetric])\ auto$

lemma $length\_filter\_conv\_size\_filter\_mset$: $length\ (filter\ P\ xs) = size\ (filter\_mset$
$P\ (mset\ xs))$
by $(induction\ xs)\ auto$

lemma *sorted_filter_less_subset_take*:
  assumes *sorted xs* **and** $i <$ *length xs*
  shows $\{\#x \in\# \mathit{mset}\ xs.\ x < xs\ !\ i\#\} \subseteq\# \mathit{mset}\ (\mathit{take}\ i\ xs)$
  using *assms*
proof (*induction xs arbitrary*: *i rule*: *list.induct*)
  case (*Cons x xs i*)
  show *?case*
  proof (*cases i*)
    case *0*
    thus *?thesis* **using** *Cons.prems* **by** (*auto simp*: *filter_mset_is_empty_iff*)
  next
    case (*Suc i'*)
    have $\{\#y \in\# \mathit{mset}\ (x\ \#\ xs).\ y < (x\ \#\ xs)\ !\ i\#\} \subseteq\# \mathit{add\_mset}\ x\ \{\#y \in\# \mathit{mset}\ xs.\ y < xs\ !\ i'\#\}$
      using *Suc Cons.prems* **by** (*auto*)
    also have $\ldots \subseteq\# \mathit{add\_mset}\ x\ (\mathit{mset}\ (\mathit{take}\ i'\ xs))$
      unfolding *mset_subset_eq_add_mset_cancel* **using** *Cons.prems Suc*
      by (*intro Cons.IH*) (*auto*)
    also have $\ldots = \mathit{mset}\ (\mathit{take}\ i\ (x\ \#\ xs))$ **by** (*simp add*: *Suc*)
    finally show *?thesis* **.**
  qed
qed *auto*

lemma *sorted_filter_greater_subset_drop*:
  assumes *sorted xs* **and** $i <$ *length xs*
  shows $\{\#x \in\# \mathit{mset}\ xs.\ x > xs\ !\ i\#\} \subseteq\# \mathit{mset}\ (\mathit{drop}\ (\mathit{Suc}\ i)\ xs)$
  using *assms*
proof (*induction xs arbitrary*: *i rule*: *list.induct*)
  case (*Cons x xs i*)
  show *?case*
  proof (*cases i*)
    case *0*
    thus *?thesis* **by** (*auto simp*: *sorted_append filter_mset_is_empty_iff*)
  next
    case (*Suc i'*)
    have $\{\#y \in\# \mathit{mset}\ (x\ \#\ xs).\ y > (x\ \#\ xs)\ !\ i\#\} \subseteq\# \{\#y \in\# \mathit{mset}\ xs.\ y > xs\ !\ i'\#\}$
      using *Suc Cons.prems* **by** (*auto simp*: *set_conv_nth*)
    also have $\ldots \subseteq\# \mathit{mset}\ (\mathit{drop}\ (\mathit{Suc}\ i')\ xs)$
      using *Cons.prems Suc* **by** (*intro Cons.IH*) (*auto*)
    also have $\ldots = \mathit{mset}\ (\mathit{drop}\ (\mathit{Suc}\ i)\ (x\ \#\ xs))$ **by** (*simp add*: *Suc*)
    finally show *?thesis* **.**
  qed
qed *auto*

49.2 Chopping a list into equally-sized bits

```
fun chop :: nat ⇒ 'a list ⇒ 'a list list where
  chop 0 _  = []
| chop _ [] = []
| chop n xs = take n xs # chop n (drop n xs)

lemmas [simp del] = chop.simps
```

This is an alternative induction rule for *chop*, which is often nicer to use.

```
lemma chop_induct' [case_names trivial reduce]:
  assumes ⋀n xs. n = 0 ∨ xs = [] ⟹ P n xs
  assumes ⋀n xs. n > 0 ⟹ xs ≠ [] ⟹ P n (drop n xs) ⟹ P n xs
  shows   P n xs
  using assms
proof induction_schema
  show wf (measure (length ∘ snd))
    by auto
qed (blast | simp)+

lemma chop_eq_Nil_iff [simp]: chop n xs = [] ⟷ n = 0 ∨ xs = []
  by (induction n xs rule: chop.induct; subst chop.simps) auto

lemma chop_0 [simp]: chop 0 xs = []
  by (simp add: chop.simps)

lemma chop_Nil [simp]: chop n [] = []
  by (cases n) (auto simp: chop.simps)

lemma chop_reduce: n > 0 ⟹ xs ≠ [] ⟹ chop n xs = take n xs # chop
n (drop n xs)
  by (cases n; cases xs) (auto simp: chop.simps)

lemma concat_chop [simp]: n > 0 ⟹ concat (chop n xs) = xs
  by (induction n xs rule: chop.induct; subst chop.simps) auto

lemma chop_elem_not_Nil [dest]: ys ∈ set (chop n xs) ⟹ ys ≠ []
  by (induction n xs rule: chop.induct; subst (asm) chop.simps)
     (auto simp: eq_commute[of []] split: if_splits)

lemma length_chop_part_le: ys ∈ set (chop n xs) ⟹ length ys ≤ n
  by (induction n xs rule: chop.induct; subst (asm) chop.simps) (auto split:
if_splits)
```

lemma *length_chop*:
  assumes $n > 0$
  shows *length* (*chop n xs*) = *nat* ⌈*length xs* / *n*⌉
proof −
  from ‹$n > 0$› **have** *real n* ∗ *length* (*chop n xs*) ≥ *length xs*
    by (*induction n xs rule*: *chop.induct*; *subst chop.simps*) (*auto simp*: *field_simps*)
  moreover from ‹$n > 0$› **have** *real n* ∗ *length* (*chop n xs*) < *length xs* + *n*
    by (*induction n xs rule*: *chop.induct*; *subst chop.simps*)
      (*auto simp*: *field_simps split*: *nat_diff_split_asm*)+
  ultimately have *length* (*chop n xs*) ≥ *length xs* / *n* **and** *length* (*chop n xs*) < *length xs* / *n* + *1*
    using *assms* **by** (*auto simp*: *field_simps*)
  thus *?thesis* **by** *linarith*
qed

lemma *sum_msets_chop*: $n > 0$ ⟹ (∑*ys*←*chop n xs*. *mset ys*) = *mset xs*
  by (*subst mset_concat* [*symmetric*]) *simp_all*

lemma *UN_sets_chop*: $n > 0$ ⟹ (⋃*ys*∈*set* (*chop n xs*). *set ys*) = *set xs*
  by (*simp only*: *set_concat* [*symmetric*] *concat_chop*)

lemma *chop_append*: *d dvd length xs* ⟹ *chop d* (*xs* @ *ys*) = *chop d xs* @ *chop d ys*
  by (*induction d xs rule*: *chop_induct'*) (*auto simp*: *chop_reduce dvd_imp_le*)

lemma *chop_replicate* [*simp*]: $d > 0$ ⟹ *chop d* (*replicate d xs*) = [*replicate d xs*]
  by (*subst chop_reduce*) *auto*

lemma *chop_replicate_dvd* [*simp*]:
  assumes *d dvd n*
  shows *chop d* (*replicate n x*) = *replicate* (*n div d*) (*replicate d x*)
proof (*cases* $d = 0$)
  case *False*
  from *assms* **obtain** *k* **where** *k*: $n = d * k$
    by *blast*
  have *chop d* (*replicate* (*d* ∗ *k*) *x*) = *replicate k* (*replicate d x*)
    using *False* **by** (*induction k*) (*auto simp*: *replicate_add chop_append*)
  thus *?thesis* **using** *False* **by** (*simp add*: *k*)
qed *auto*

lemma *chop_concat*:
  assumes $\forall$ *xs*$\in$*set xss*. *length xs* = *d* **and** *d* > *0*
  shows   *chop d* (*concat xss*) = *xss*
  using *assms*
proof (*induction xss*)
  case (*Cons xs xss*)
  have *chop d* (*concat* (*xs* # *xss*)) = *chop d* (*xs* @ *concat xss*)
    by *simp*
  also have . . . = *chop d xs* @ *chop d* (*concat xss*)
    using *Cons.prems* **by** (*intro chop_append*) *auto*
  also have *chop d xs* = [*xs*]
    using *Cons.prems* **by** (*subst chop_reduce*) *auto*
  also have *chop d* (*concat xss*) = *xss*
    using *Cons.prems* **by** (*intro Cons.IH*) *auto*
  finally show *?case* **by** *simp*
qed *auto*

49.3 Selection

definition *select* :: *nat* $\Rightarrow$ (*'a* :: *linorder*) *list* $\Rightarrow$ *'a* **where**
  select k xs = *sort xs* ! *k*

lemma *select_0*: *xs* $\neq$ [] $\Longrightarrow$ *select 0 xs* = *Min* (*set xs*)
  by (*simp add*: *hd_sort select_def flip*: *hd_conv_nth*)

lemma *select_mset_cong*: *mset xs* = *mset ys* $\Longrightarrow$ *select k xs* = *select k ys*
  using *sort_mset_cong*[*of xs ys*] **unfolding** *select_def* **by** *auto*

lemma *select_in_set* [*intro*,*simp*]:
  assumes *k* < *length xs*
  shows   *select k xs* $\in$ *set xs*
proof −
  from *assms* **have** *sort xs* ! *k* $\in$ *set* (*sort xs*) **by** (*intro nth_mem*) *auto*
  also have *set* (*sort xs*) = *set xs* **by** *simp*
  finally show *?thesis* **by** (*simp add*: *select_def*)
qed

lemma
  assumes *n* < *length xs*
  shows   *size_less_than_select*: *size* {#*y* $\in$# *mset xs*. *y* < *select n xs*#}
$\leq$ *n*
   and   *size_greater_than_select*: *size* {#*y* $\in$# *mset xs*. *y* > *select n xs*#}
< *length xs* − *n*
proof −

have *size* {#*y* ∈# *mset* (*sort xs*). *y* < *select n xs*#} ≤ *size* (*mset* (*take n* (*sort xs*)))
  unfolding *select_def* **using** *assms*
  by (*intro size_mset_mono sorted_filter_less_subset_take*) *auto*
thus *size* {#*y* ∈# *mset xs*. *y* < *select n xs*#} ≤ *n*
  by *simp*
have *size* {#*y* ∈# *mset* (*sort xs*). *y* > *select n xs*#} ≤ *size* (*mset* (*drop* (*Suc n*) (*sort xs*)))
  unfolding *select_def* **using** *assms*
  by (*intro size_mset_mono sorted_filter_greater_subset_drop*) *auto*
thus *size* {#*y* ∈# *mset xs*. *y* > *select n xs*#} < *length xs* − *n*
  using *assms* **by** *simp*
qed

49.4 The designated median of a list

definition *median* **where** *median xs* = *select* ((*length xs* − *1*) *div 2*) *xs*

lemma *median_in_set* [*intro*, *simp*]:
assumes *xs* ≠ []
shows *median xs* ∈ *set xs*
proof −
from *assms* **have** *length xs* > *0* **by** *auto*
hence (*length xs* − *1*) *div 2* < *length xs* **by** *linarith*
thus *?thesis* **by** (*simp add*: *median_def*)
qed

lemma *size_less_than_median*: *size* {#*y* ∈# *mset xs*. *y* < *median xs*#} ≤ (*length xs* − *1*) *div 2*
proof (*cases xs* = [])
case *False*
hence *length xs* > *0*
  by *auto*
hence *less*: (*length xs* − *1*) *div 2* < *length xs*
  by *linarith*
show *size* {#*y* ∈# *mset xs*. *y* < *median xs*#} ≤ (*length xs* − *1*) *div 2*
  using *size_less_than_select*[*OF less*] **by** (*simp add*: *median_def*)
qed *auto*

lemma *size_greater_than_median*: *size* {#*y* ∈# *mset xs*. *y* > *median xs*#} ≤ *length xs div 2*
proof (*cases xs* = [])
case *False*
hence *length xs* > *0*

```
    by auto
  hence less: (length xs − 1) div 2 < length xs
    by linarith
  have size {#y ∈# mset xs. y > median xs#} < length xs − (length xs −
1) div 2
    using size_greater_than_select[OF less] by (simp add: median_def)
  also have ... = length xs div 2 + 1
    using ‹length xs > 0› by linarith
  finally show size {#y ∈# mset xs. y > median xs#} ≤ length xs div 2
    by simp
qed auto

lemmas median_props = size_less_than_median size_greater_than_median
```

49.5 A recurrence for selection

```
definition partition3 :: 'a ⇒ 'a :: linorder list ⇒ 'a list × 'a list × 'a list
where
  partition3 x xs = (filter (λy. y < x) xs, filter (λy. y = x) xs, filter (λy. y
> x) xs)

lemma partition3_code [code]:
  partition3 x [] = ([], [], [])
  partition3 x (y # ys) =
    (case partition3 x ys of (ls, es, gs) ⇒
       if y < x then (y # ls, es, gs) else if x = y then (ls, y # es, gs) else
(ls, es, y # gs))
  by (auto simp: partition3_def)

lemma sort_append:
  assumes ∀x∈set xs. ∀y∈set ys. x ≤ y
  shows   sort (xs @ ys) = sort xs @ sort ys
  using assms by (intro properties_for_sort) (auto simp: sorted_append)

lemma select_append:
  assumes ∀y∈set ys. ∀z∈set zs. y ≤ z
  shows   k < length ys ⟹ select k (ys @ zs) = select k ys
    and   k ∈ {length ys..<length ys + length zs} ⟹
            select k (ys @ zs) = select (k − length ys) zs
  using assms by (simp_all add: select_def sort_append nth_append)

lemma select_append':
  assumes ∀y∈set ys. ∀z∈set zs. y ≤ z and k < length ys + length zs
  shows   select k (ys @ zs) = (if k < length ys then select k ys else select
(k − length ys) zs)
  using assms by (auto intro!: select_append)

theorem select_rec_partition:
  assumes k < length xs
  shows select k xs = (
          let (ls, es, gs) = partition3 x xs
          in
            if k < length ls then select k ls
            else if k < length ls + length es then x
            else select (k − length ls − length es) gs
        ) (is _ = ?rhs)
proof −
  define ls es gs where ls = filter (λy. y < x) xs and es = filter (λy. y =
x) xs
                   and gs = filter (λy. y > x) xs
  define nl ne where [simp]: nl = length ls ne = length es
  have mset_eq: mset xs = mset ls + mset es + mset gs
    unfolding ls_def es_def gs_def by (induction xs) auto
  have length_eq: length xs = length ls + length es + length gs
    unfolding ls_def es_def gs_def
    using [[simp_depth_limit = 1]] by (induction xs) auto
  have [simp]: select i es = x if i < length es for i
  proof −
    have select i es ∈ set (sort es) unfolding select_def
      using that by (intro nth_mem) auto
    thus ?thesis
      by (auto simp: es_def)
  qed

  have select k xs = select k (ls @ (es @ gs))
    by (intro select_mset_cong) (simp_all add: mset_eq)
  also have ... = (if k < nl then select k ls else select (k − nl) (es @ gs))
    unfolding nl_ne_def using assms
    by (intro select_append') (auto simp: ls_def es_def gs_def length_eq)
  also have ... = (if k < nl then select k ls else if k < nl + ne then x
                   else select (k − nl − ne) gs)
  proof (rule if_cong)
    assume ¬k < nl
    have select (k − nl) (es @ gs) =
                 (if k − nl < ne then select (k − nl) es else select (k − nl −
ne) gs)
      unfolding nl_ne_def using assms ‹¬k < nl›
      by (intro select_append') (auto simp: ls_def es_def gs_def length_eq)
    also have ... = (if k < nl + ne then x else select (k − nl − ne) gs)
      using ‹¬k < nl› by auto
    finally show select (k − nl) (es @ gs) = ... .
  qed simp_all
  also have ... = ?rhs
    by (simp add: partition3_def ls_def es_def gs_def)
  finally show ?thesis .
qed
```

49.6 The size of the lists in the recursive calls

We now derive an upper bound for the number of elements of a list that are smaller (resp. bigger) than the median of medians with chopping size 5. To avoid having to do the same proof twice, we do it generically for an operation ≺ that we will later instantiate with either < or >.

```
context
  fixes xs :: 'a :: linorder list
  fixes M defines M ≡ median (map median (chop 5 xs))
begin


lemma size_median_of_medians_aux:
  fixes R :: 'a :: linorder ⇒ 'a ⇒ bool (infix ≺ 50)
  assumes R: R ∈ {(<), (>)}
  shows size {#y ∈# mset xs. y ≺ M#} ≤ nat ⌈0.7 ∗ length xs + 3⌉
proof −
  define n and m where [simp]: n = length xs and m = length (chop 5
xs)
```

We define an abbreviation for the multiset of all the chopped-up groups:

We then split that multiset into those groups whose medians is less than M and the rest.

```
  define Y_small (Y≺) where Y≺ = filter_mset (λys. median ys ≺ M)
(mset (chop 5 xs))
  define Y_big (Y⪰) where Y⪰ = filter_mset (λys. ¬(median ys ≺ M))
(mset (chop 5 xs))
  have m = size (mset (chop 5 xs)) by (simp add: m_def)
  also have mset (chop 5 xs) = Y≺ + Y⪰ unfolding Y_small_def Y_big_def
    by (rule multiset_partition)
  finally have m_eq: m = size Y≺ + size Y⪰ by simp
```

At most half of the lists have a median that is smaller than the median of medians:

```
  have size Y≺ = size (image_mset median Y≺) by simp
```

also have *image_mset median* $Y_{\prec}$ = {#*y* ∈# *mset* (*map median* (*chop 5 xs*)). *y* ≺ *M*#}
unfolding *Y_small_def* **by** (*subst filter_mset_image_mset* [*symmetric*]) *simp_all*
also have *size* ... ≤ (*length* (*map median* (*chop 5 xs*))) *div 2*
unfolding *M_def* **using** *median_props*[*of map median* (*chop 5 xs*)] *R* **by** *auto*
also have ... = *m div 2* **by** (*simp add*: *m_def*)
finally have *size_Y_small*: *size* $Y_{\prec}$ ≤ *m div 2* **.**

We estimate the number of elements less than *M* by grouping them into elements coming from $Y_{\prec}$ and elements coming from $Y_{\succeq}$:

have {#*y* ∈# *mset xs*. *y* ≺ *M*#} = {#*y* ∈# (∑ *ys*←*chop 5 xs*. *mset ys*). *y* ≺ *M*#}
by (*subst sum_msets_chop*) *simp_all*
also have ... = (∑ *ys*←*chop 5 xs*. {#*y* ∈# *mset ys*. *y* ≺ *M*#})
by (*subst filter_mset_sum_list*) (*simp add*: *o_def*)
also have ... = (∑ *ys*∈#*mset* (*chop 5 xs*). {#*y* ∈# *mset ys*. *y* ≺ *M*#})
by (*subst sum_mset_sum_list* [*symmetric*]) *simp_all*
also have *mset* (*chop 5 xs*) = $Y_{\prec}$ + $Y_{\succeq}$
by (*simp add*: *Y_small_def Y_big_def not_le*)
also have (∑ *ys*∈#.... {#*y* ∈# *mset ys*. *y* ≺ *M*#}) =
(∑ *ys*∈#$Y_{\prec}$. {#*y* ∈# *mset ys*. *y* ≺ *M*#}) + (∑ *ys*∈#$Y_{\succeq}$. {#*y* ∈# *mset ys*. *y* ≺ *M*#})
by *simp*

Next, we overapproximate the elements contributed by $Y_{\prec}$: instead of those elements that are smaller than the median, we take *all* the elements of each group. For the elements contributed by $Y_{\succeq}$, we overapproximate by taking all those that are less than their median instead of only those that are less than *M*.

also have ... ⊆# (∑ *ys*∈#$Y_{\prec}$. *mset ys*) + (∑ *ys*∈#$Y_{\succeq}$. {#*y* ∈# *mset ys*. *y* ≺ *median ys*#})
using *R*
by (*intro subset_mset.add_mono sum_mset_mset_mono mset_filter_mono*) (*auto simp*: *Y_big_def*)
finally have *size* {# *y* ∈# *mset xs*. *y* ≺ *M*#} ≤ *size* ...
by (*rule size_mset_mono*)
hence *size* {# *y* ∈# *mset xs*. *y* ≺ *M*#} ≤
(∑ *ys*∈#$Y_{\prec}$. *length ys*) + (∑ *ys*∈#$Y_{\succeq}$. *size* {#*y* ∈# *mset ys*. *y* ≺ *median ys*#})
by (*simp add*: *size_mset_sum_mset_distrib multiset.map_comp o_def*)

Next, we further overapproximate the first sum by noting that each group has at most size 5.

```
  also have (∑ys∈#Y≺. length ys) ≤ (∑ys∈#Y≺. 5)
   by (intro sum_mset_mono) (auto simp: Y_small_def length_chop_part_le)
  also have ... = 5 ∗ size Y≺ by simp
```

Next, we note that each group in $Y_\succeq$ can have at most 2 elements that are smaller than its median.

```
  also have (∑ys∈#Y⪰. size {#y ∈# mset ys. y ≺ median ys#}) ≤
            (∑ys∈#Y⪰. length ys div 2)
  proof (intro sum_mset_mono, goal_cases)
    fix ys assume ys ∈# Y⪰
    hence ys ≠ []
      by (auto simp: Y_big_def)
    thus size {#y ∈# mset ys. y ≺ median ys#} ≤ length ys div 2
      using R median_props[of ys] by auto
  qed
  also have ... ≤ (∑ys∈#Y⪰. 2)
    by (intro sum_mset_mono div_le_mono diff_le_mono)
       (auto simp: Y_big_def dest: length_chop_part_le)
  also have ... = 2 ∗ size Y⪰ by simp
```

Simplifying gives us the main result.

```
  also have 5 ∗ size Y≺ + 2 ∗ size Y⪰ = 2 ∗ m + 3 ∗ size Y≺
    by (simp add: m_eq)
  also have ... ≤ 3.5 ∗ m
    using ‹size Y≺ ≤ m div 2› by linarith
  also have ... = 3.5 ∗ ⌈n / 5⌉
    by (simp add: m_def length_chop)
  also have ... ≤ 0.7 ∗ n + 3.5
    by linarith
  finally have size {#y ∈# mset xs. y ≺ M#} ≤ 0.7 ∗ n + 3.5
    by simp
  thus size {#y ∈# mset xs. y ≺ M#} ≤ nat ⌈0.7 ∗ n + 3⌉
    by linarith
qed

lemma size_less_than_median_of_medians:
  size {#y ∈# mset xs. y < M#} ≤ nat ⌈0.7 ∗ length xs + 3⌉
  using size_median_of_medians_aux[of (<)] by simp

lemma size_greater_than_median_of_medians:
  size {#y ∈# mset xs. y > M#} ≤ nat ⌈0.7 ∗ length xs + 3⌉
  using size_median_of_medians_aux[of (>)] by simp

end
```

49.7 Efficient algorithm

We handle the base cases and computing the median for the chopped-up sublists of size 5 using the naive selection algorithm where we sort the list using insertion sort.

```
definition slow_select where
  slow_select k xs = isort xs ! k

definition slow_median where
  slow_median xs = slow_select ((length xs − 1) div 2) xs

lemma slow_select_correct: slow_select k xs = select k xs
  by (simp add: slow_select_def select_def isort_correct)

lemma slow_median_correct: slow_median xs = median xs
  by (simp add: median_def slow_median_def slow_select_correct)
```

The definition of the selection algorithm is complicated somewhat by the fact that its termination is contingent on its correctness: if the first recursive call were to return an element for x that is e.g. smaller than all list elements, the algorithm would not terminate.

Therefore, we first prove partial correctness, then termination, and then combine the two to obtain total correctness.

```
function mom_select where
  mom_select k xs = (
    if length xs ≤ 20 then
      slow_select k xs
    else
        let M = mom_select (((length xs + 4) div 5 − 1) div 2) (map
slow_median (chop 5 xs));
          (ls, es, gs) = partition3 M xs
      in
        if k < length ls then mom_select k ls
        else if k < length ls + length es then M
        else mom_select (k − length ls − length es) gs
    )
  by auto
```

If *mom_select* terminates, it agrees with *select*:

```
lemma mom_select_correct_aux:
  assumes mom_select_dom (k, xs) and k < length xs
  shows   mom_select k xs = select k xs
  using assms
proof (induction rule: mom_select.pinduct)
```

```
  case (1 k xs)
  show mom_select k xs = select k xs
  proof (cases length xs ≤ 20)
    case True
    thus mom_select k xs = select k xs using 1.prems 1.hyps
     by (subst mom_select.psimps) (auto simp: select_def slow_select_correct)
  next
    case False
    define x where
     x = mom_select (((length xs + 4) div 5 − 1) div 2) (map slow_median
(chop 5 xs))
    define ls es gs where ls = filter (λy. y < x) xs and es = filter (λy. y
= x) xs
                      and gs = filter (λy. y > x) xs
    define nl ne where nl = length ls and ne = length es
    note defs = nl_def ne_def x_def ls_def es_def gs_def
    have tw: (ls, es, gs) = partition3 x xs
      unfolding partition3_def defs One_nat_def ..
    have length_eq: length xs = nl + ne + length gs
      unfolding nl_def ne_def ls_def es_def gs_def
      using [[simp_depth_limit = 1]] by (induction xs) auto
    note IH = 1.IH(2,3)[OF False x_def tw refl refl]

    have mom_select k xs = (if k < nl then mom_select k ls else if k < nl
+ ne then x
                                else mom_select (k − nl − ne) gs) using 1.hyps
False
      by (subst mom_select.psimps) (simp_all add: partition3_def flip: defs
One_nat_def)
    also have ... = (if k < nl then select k ls else if k < nl + ne then x
                         else select (k − nl − ne) gs)
     using IH length_eq 1.prems by (simp add: ls_def es_def gs_def nl_def
ne_def)
    also have ... = select k xs using ‹k < length xs›
      by (subst (3) select_rec_partition[of _ _ x]) (simp_all add: nl_def
ne_def flip: tw)
    finally show mom_select k xs = select k xs .
  qed
qed
```

mom_select indeed terminates for all inputs:

```
lemma mom_select_termination: All mom_select_dom
proof (relation measure (length ∘ snd); (safe)?)
  fix k :: nat and xs :: 'a list
```

assume $\neg$ *length xs* $\leq$ *20*
thus $((((\mathit{length\ xs} + 4)\ \mathit{div}\ 5 - 1)\ \mathit{div}\ 2,\ \mathit{map\ slow\_median}\ (\mathit{chop}\ 5\ \mathit{xs})),\ k,\ \mathit{xs})$
$\in \mathit{measure}\ (\mathit{length} \circ \mathit{snd})$
by (*auto simp*: *length_chop nat_less_iff ceiling_less_iff*)
next
fix *k* :: *nat* **and** *xs ls es gs* :: *'a list*
define *x* **where** $x = \mathit{mom\_select}\ (((\mathit{length\ xs} + 4)\ \mathit{div}\ 5 - 1)\ \mathit{div}\ 2)\ (\mathit{map\ slow\_median}\ (\mathit{chop}\ 5\ \mathit{xs}))$
assume *A*: $\neg$ *length xs* $\leq$ *20*
$(\mathit{ls},\ \mathit{es},\ \mathit{gs}) = \mathit{partition3}\ x\ \mathit{xs}$
$\mathit{mom\_select\_dom}\ (((\mathit{length\ xs} + 4)\ \mathit{div}\ 5 - 1)\ \mathit{div}\ 2,$
$\mathit{map\ slow\_median}\ (\mathit{chop}\ 5\ \mathit{xs}))$
have *less*: $((\mathit{length\ xs} + 4)\ \mathit{div}\ 5 - 1)\ \mathit{div}\ 2 < \mathit{nat}\ \lceil \mathit{length\ xs} / 5 \rceil$
using *A(1)* **by** *linarith*

For termination, it suffices to prove that *x* is in the list.

have $x = \mathit{select}\ (((\mathit{length\ xs} + 4)\ \mathit{div}\ 5 - 1)\ \mathit{div}\ 2)\ (\mathit{map\ slow\_median}\ (\mathit{chop}\ 5\ \mathit{xs}))$
using *less* **unfolding** *x_def* **by** (*intro mom_select_correct_aux A*) (*auto simp*: *length_chop*)
also have ... $\in$ *set* (*map slow_median* (*chop 5 xs*))
using *less* **by** (*intro select_in_set*) (*simp_all add*: *length_chop*)
also have ... $\subseteq$ *set xs*
unfolding *set_map*
proof *safe*
fix *ys* **assume** *ys*: *ys* $\in$ *set* (*chop 5 xs*)
hence *median ys* $\in$ *set ys*
by *auto*
also have *set ys* $\subseteq \bigcup(\mathit{set}$ ` $\mathit{set}\ (\mathit{chop}\ 5\ \mathit{xs}))$
using *ys* **by** *blast*
also have ... = *set xs*
by (*rule UN_sets_chop*) *simp_all*
finally show *slow_median ys* $\in$ *set xs*
by (*simp add*: *slow_median_correct*)
qed
finally have *x* $\in$ *set xs* **.**
thus $((k,\ \mathit{ls}),\ k,\ \mathit{xs}) \in \mathit{measure}\ (\mathit{length} \circ \mathit{snd})$
and $((k - \mathit{length\ ls} - \mathit{length\ es},\ \mathit{gs}),\ k,\ \mathit{xs}) \in \mathit{measure}\ (\mathit{length} \circ \mathit{snd})$
using *A(1,2)* **by** (*auto simp*: *partition3_def intro*!: *length_filter_less*[*of x*])
qed

termination *mom_select* **by** (*rule mom_select_termination*)

lemmas [*simp del*] = *mom_select.simps*

lemma *mom_select_correct*: *k* < *length xs* ⟹ *mom_select k xs* = *select k xs*
  using *mom_select_correct_aux* **and** *mom_select_termination* **by** *blast*

49.8 Running time analysis

fun *T_partition3* :: *'a* ⇒ *'a list* ⇒ *nat* **where**
  T_partition3 x [] = *1*
| *T_partition3 x* (*y* # *ys*) = *T_partition3 x ys* + *1*

lemma *T_partition3_eq*: *T_partition3 x xs* = *length xs* + *1*
  by (*induction x xs rule*: *T_partition3.induct*) *auto*

definition *T_slow_select* :: *nat* ⇒ *'a* :: *linorder list* ⇒ *nat* **where**
  T_slow_select k xs = *T_isort xs* + *T_nth* (*isort xs*) *k* + *1*

definition *T_slow_median* :: *'a* :: *linorder list* ⇒ *nat* **where**
  T_slow_median xs = *T_slow_select* ((*length xs* − *1*) *div 2*) *xs* + *1*

lemma *T_slow_select_le*: *T_slow_select k xs* ≤ *length xs* ^ *2* + *3* ∗ *length xs* + *3*
proof −
  have *T_slow_select k xs* ≤ (*length xs* + *1*)^2 + (*length* (*isort xs*) + *1*) + *1*
    unfolding *T_slow_select_def*
    by (*intro add_mono T_isort_length*) (*auto simp*: *T_nth_eq*)
  also have ... = *length xs* ^ *2* + *3* ∗ *length xs* + *3*
    by (*simp add*: *isort_correct algebra_simps power2_eq_square*)
  finally show *?thesis* **.**
qed

lemma *T_slow_median_le*: *T_slow_median xs* ≤ *length xs* ^ *2* + *3* ∗ *length xs* + *4*
  unfolding *T_slow_median_def* **using** *T_slow_select_le*[*of* (*length xs* − *1*) *div 2 xs*] **by** *simp*

fun *T_chop* :: *nat* ⇒ *'a list* ⇒ *nat* **where**
  T_chop 0 _ = *1*
| *T_chop _* [] = *1*
| *T_chop n xs* = *T_take n xs* + *T_drop n xs* + *T_chop n* (*drop n xs*)

lemmas [*simp del*] = *T_chop.simps*

lemma *T_chop_Nil* [*simp*]: *T_chop d* [] = *1*
  by (*cases d*) (*auto simp*: *T_chop.simps*)

lemma *T_chop_0* [*simp*]: *T_chop 0 xs* = *1*
  by (*auto simp*: *T_chop.simps*)

lemma *T_chop_reduce*:
  $n > 0 \Longrightarrow xs \neq [] \Longrightarrow$ *T_chop n xs* = *T_take n xs* + *T_drop n xs* + *T_chop n* (*drop n xs*)
  by (*cases n*; *cases xs*) (*auto simp*: *T_chop.simps*)

lemma *T_chop_le*: *T_chop d xs* $\leq$ *5* $*$ *length xs* + *1*
  by (*induction d xs rule*: *T_chop.induct*) (*auto simp*: *T_chop_reduce T_take_eq T_drop_eq*)

The option *domintros* here allows us to explicitly reason about where the function does and does not terminate. With this, we can skip the termination proof this time because we can reuse the one for *mom_select*.

function (*domintros*) *T_mom_select* :: *nat* ⇒ *'a* :: *linorder list* ⇒ *nat* **where**

```
  T_mom_select k xs = (
    if length xs ≤ 20 then
      T_slow_select k xs
    else
      let xss = chop 5 xs;
          ms = map slow_median xss;
          idx = (((length xs + 4) div 5 − 1) div 2);
          x = mom_select idx ms;
          (ls, es, gs) = partition3 x xs;
          nl = length ls;
          ne = length es
      in
        (if k < nl then T_mom_select k ls
         else if k < nl + ne then 0
         else T_mom_select (k − nl − ne) gs) +
         T_mom_select idx ms + T_chop 5 xs + T_map T_slow_median xss +
        T_partition3 x xs + T_length ls + T_length es + 1
    )
  by auto
```

```
termination T_mom_select
proof (rule allI, safe)
  fix k :: nat and xs :: 'a :: linorder list
  have mom_select_dom (k, xs)
    using mom_select_termination by blast
  thus T_mom_select_dom (k, xs)
    by (induction k xs rule: mom_select.pinduct)
       (rule T_mom_select.domintros, simp_all)
qed

lemmas [simp del] = T_mom_select.simps


function T'_mom_select :: nat ⇒ nat where
  T'_mom_select n =
     (if n ≤ 20 then
        463
      else
        T'_mom_select (nat ⌈0.2∗n⌉) + T'_mom_select (nat ⌈0.7∗n+3⌉)
+ 17 ∗ n + 50)
  by force+
termination by (relation measure id; simp; linarith)

lemmas [simp del] = T'_mom_select.simps


lemma T'_mom_select_ge: T'_mom_select n ≥ 463
 by (induction n rule: T'_mom_select.induct; subst T'_mom_select.simps)
auto

lemma T'_mom_select_mono:
  m ≤ n ⟹ T'_mom_select m ≤ T'_mom_select n
proof (induction n arbitrary: m rule: less_induct)
  case (less n m)
  show ?case
  proof (cases m ≤ 20)
    case True
    hence T'_mom_select m = 463
      by (subst T'_mom_select.simps) auto
    also have ... ≤ T'_mom_select n
      by (rule T'_mom_select_ge)
    finally show ?thesis .
  next
    case False
```

hence *T′_mom_select m =*
T′_mom_select (*nat* ⌈*0.2*∗*m*⌉) + *T′_mom_select* (*nat* ⌈*0.7*∗*m* + *3*⌉) + *17* ∗ *m* + *50*
by (*subst T′_mom_select.simps*) *auto*
also have . . . ≤ *T′_mom_select* (*nat* ⌈*0.2*∗*n*⌉) + *T′_mom_select* (*nat* ⌈*0.7*∗*n* + *3*⌉) + *17* ∗ *n* + *50*
using ‹*m* ≤ *n*› **and** *False* **by** (*intro add_mono less.IH*; *linarith*)
also have . . . = *T′_mom_select n*
using ‹*m* ≤ *n*› **and** *False* **by** (*subst T′_mom_select.simps*) *auto*
finally show *?thesis* **.**
qed
qed

lemma *T_mom_select_le_aux*: *T_mom_select k xs* ≤ *T′_mom_select* (*length xs*)
proof (*induction k xs rule*: *T_mom_select.induct*)
case (*1 k xs*)
define *n* **where** [*simp*]: *n* = *length xs*
define *x* **where**
x = *mom_select* (((*length xs* + *4*) *div 5* − *1*) *div 2*) (*map slow_median* (*chop 5 xs*))
define *ls es gs* **where** *ls* = *filter* (λ*y*. *y* < *x*) *xs* **and** *es* = *filter* (λ*y*. *y* = *x*) *xs*
and *gs* = *filter* (λ*y*. *y* > *x*) *xs*
define *nl ne* **where** *nl* = *length ls* **and** *ne* = *length es*
note *defs* = *nl_def ne_def x_def ls_def es_def gs_def*
have *tw*: (*ls*, *es*, *gs*) = *partition3 x xs*
unfolding *partition3_def defs One_nat_def* **..**
note *IH* = *1.IH*(*1*,*2*,*3*)[*OF _ refl refl refl x_def tw refl refl refl refl*]

show *?case*
proof (*cases length xs* ≤ *20*)
case *True* — base case
hence *T_mom_select k xs* ≤ (*length xs*)^2 + *3* ∗ *length xs* + *3*
using *T_slow_select_le*[*of k xs*] **by** (*subst T_mom_select.simps*) *auto*
also have . . . ≤ *20*^{2} + *3* ∗ *20* + *3*
using *True* **by** (*intro add_mono power_mono*) *auto*
also have . . . ≤ *463*
by *simp*
also have . . . = *T′_mom_select* (*length xs*)
using *True* **by** (*simp add*: *T′_mom_select.simps*)
finally show *?thesis* **by** *simp*
next
case *False* — recursive case

```
  have ((n + 4) div 5 − 1) div 2 < nat ⌈n / 5⌉
    using False unfolding n_def by linarith
  hence x = select (((n + 4) div 5 − 1) div 2) (map slow_median (chop 5 xs))
      unfolding x_def n_def by (intro mom_select_correct) (auto simp: length_chop)
  also have ((n + 4) div 5 − 1) div 2 = (nat ⌈n / 5⌉ − 1) div 2
    by linarith
  also have select ... (map slow_median (chop 5 xs)) = median (map slow_median (chop 5 xs))
    by (auto simp: median_def length_chop)
  finally have x_eq: x = median (map slow_median (chop 5 xs)) .
```

The cost of computing the medians of all the subgroups:

```
  define T_ms where T_ms = T_map T_slow_median (chop 5 xs)
  have T_ms ≤ 9 ∗ n + 45
  proof −
    have T_ms = (∑ys←chop 5 xs. T_slow_median ys) + length (chop 5 xs) + 1
      by (simp add: T_ms_def T_map_eq)
    also have (∑ys←chop 5 xs. T_slow_median ys) ≤ (∑ys←chop 5 xs. 44)
    proof (intro sum_list_mono)
      fix ys assume ys ∈ set (chop 5 xs)
      hence length ys ≤ 5
        using length_chop_part_le by blast
      have T_slow_median ys ≤ (length ys) ^ 2 + 3 ∗ length ys + 4
        by (rule T_slow_median_le)
      also have ... ≤ 5 ^ 2 + 3 ∗ 5 + 4
        using ‹length ys ≤ 5› by (intro add_mono power_mono) auto
      finally show T_slow_median ys ≤ 44 by simp
    qed
    also have (∑ys←chop 5 xs. 44) + length (chop 5 xs) + 1 =
                45 ∗ nat ⌈real n / 5⌉ + 1
      by (simp add: map_replicate_const length_chop)
    also have ... ≤ 9 ∗ n + 45
      by linarith
    finally show T_ms ≤ 9 ∗ n + 45 by simp
  qed
```

The cost of the first recursive call (to compute the median of medians):

```
  define T_rec1 where
    T_rec1 = T_mom_select (((length xs + 4) div 5 − 1) div 2) (map slow_median (chop 5 xs))
```

have $T\_rec1 \leq T'\_mom\_select$ (*length* (*map slow_median* (*chop 5 xs*)))
  using *False* **unfolding** *T_rec1_def* **by** (*intro IH*(*3*)) *auto*
hence $T\_rec1 \leq T'\_mom\_select$ (*nat* $\lceil 0.2 * n \rceil$)
  by (*simp add*: *length_chop*)

The cost of the second recursive call (to compute the final result):

define *T_rec2* **where** *T_rec2* = (*if* $k < nl$ *then T_mom_select k ls*
    else if $k < nl + ne$ *then 0*
    else T_mom_select ($k - nl - ne$) *gs*)
consider $k < nl \mid k \in \{nl..{<}nl{+}ne\} \mid k \geq nl{+}ne$
  by *force*
hence $T\_rec2 \leq T'\_mom\_select$ (*nat* $\lceil 0.7 * n + 3 \rceil$)
proof *cases*
  assume $k < nl$
  hence *T_rec2* = *T_mom_select k ls*
    by (*simp add*: *T_rec2_def*)
  also have ... $\leq T'\_mom\_select$ (*length ls*)
    by (*rule IH*(*1*)) (*use* ‹$k < nl$› *False* **in** ‹*auto simp*: *defs*›)
  also have *length ls* $\leq$ *nat* $\lceil 0.7 * n + 3 \rceil$
    unfolding *ls_def* **using** *size_less_than_median_of_medians*[*of xs*]
  by (*auto simp*: *length_filter_conv_size_filter_mset slow_median_correct*[*abs_def*] *x_eq*)
  hence T'_mom_select (*length ls*) $\leq T'\_mom\_select$ (*nat* $\lceil 0.7 * n + 3 \rceil$)
    by (*rule T'_mom_select_mono*)
  finally show *?thesis* **.**
next
  assume $k \in \{nl..{<}nl + ne\}$
  hence *T_rec2* = *0*
    by (*simp add*: *T_rec2_def*)
  thus *?thesis*
    using *T'_mom_select_ge*[*of nat* $\lceil 0.7 * n + 3 \rceil$] **by** *simp*
next
  assume $k \geq nl + ne$
  hence *T_rec2* = *T_mom_select* ($k - nl - ne$) *gs*
    by (*simp add*: *T_rec2_def*)
  also have ... $\leq T'\_mom\_select$ (*length gs*)
    unfolding *nl_def ne_def* **by** (*rule IH*(*2*)) (*use* ‹$k \geq nl + ne$› *False* **in** ‹*auto simp*: *defs*›)
  also have *length gs* $\leq$ *nat* $\lceil 0.7 * n + 3 \rceil$
    unfolding *gs_def* **using** *size_greater_than_median_of_medians*[*of xs*]
  by (*auto simp*: *length_filter_conv_size_filter_mset slow_median_correct*[*abs_def*]

x_eq)
    hence *T′_mom_select* (*length gs*) ≤ *T′_mom_select* (*nat* ⌈*0.7* ∗ *n* + *3*⌉)
      by (*rule T′_mom_select_mono*)
    finally show *?thesis* **.**
  qed

Now for the final inequality chain:

  have *T_mom_select k xs* = *T_rec2* + *T_rec1* + *T_ms* + *n* + *nl* + *ne* + *T_chop 5 xs* + *4* **using** *False*
    by (*subst T_mom_select.simps*, *unfold Let_def tw* [*symmetric*] *defs* [*symmetric*])
      (*simp_all add*: *nl_def ne_def T_rec1_def T_rec2_def T_partition3_eq T_length_eq T_ms_def*)
  also have *nl* ≤ *n* **by** (*simp add*: *nl_def ls_def*)
  also have *ne* ≤ *n* **by** (*simp add*: *ne_def es_def*)
  also note ‹*T_ms* ≤ *9* ∗ *n* + *45*›
  also have *T_chop 5 xs* ≤ *5* ∗ *n* + *1*
    using *T_chop_le*[*of 5 xs*] **by** *simp*
  also note ‹*T_rec1* ≤ *T′_mom_select* (*nat* ⌈*0.2*∗*n*⌉)›
  also note ‹*T_rec2* ≤ *T′_mom_select* (*nat* ⌈*0.7*∗*n* + *3*⌉)›
  finally have *T_mom_select k xs* ≤
      T′_mom_select (*nat* ⌈*0.7*∗*n* + *3*⌉) + *T′_mom_select* (*nat* ⌈*0.2*∗*n*⌉) + *17* ∗ *n* + *50*
    by *simp*
  also have ... = *T′_mom_select n*
    using *False* **by** (*subst T′_mom_select.simps*) *auto*
  finally show *?thesis* **by** *simp*
 qed
qed

49.9 Akra–Bazzi Light

lemma *akra_bazzi_light_aux1*:
 fixes *a b* :: *real* **and** *n n0* :: *nat*
 assumes *ab*: *a* > *0 a* < *1 n* > *n0*
 assumes *n0* ≥ (*max 0 b* + *1*) / (*1* − *a*)
 shows *nat* ⌈*a*∗*n*+*b*⌉ < *n*
proof −
 have *a* ∗ *real n* + *max 0 b* ≥ *0*
  using *ab* **by** *simp*
 hence *real* (*nat* ⌈*a*∗*n*+*b*⌉) ≤ *a* ∗ *n* + *max 0 b* + *1*
  by *linarith*
 also {

```
    have n0 ≥ (max 0 b + 1) / (1 − a)
      by fact
    also have ... < real n
      using assms by simp
    finally have a ∗ real n + max 0 b + 1 < real n
      using ab by (simp add: field_simps)
  }
  finally show nat ⌈a∗n+b⌉ < n
    using ‹n > n0› by linarith
qed

lemma akra_bazzi_light_aux2:
  fixes f :: nat ⇒ real
  fixes n_0 :: nat and a b c d :: real and C1 C2 C_1 C_2 :: real
  assumes bounds: a > 0 c > 0 a + c < 1 C_1 ≥ 0
  assumes rec: ∀ n>n_0. f n = f (nat ⌈a∗n+b⌉) + f (nat ⌈c∗n+d⌉) + C_1 ∗
n + C_2
  assumes ineqs: n_0 > (max 0 b + max 0 d + 2) / (1 − a − c)
               C_3 ≥ C_1 / (1 − a − c)
               C_3 ≥ (C_1 ∗ n_0 + C_2 + C_4) / ((1 − a − c) ∗ n_0 − max 0 b
− max 0 d − 2)
               ∀ n≤n_0. f n ≤ C_4
  shows  f n ≤ C_3 ∗ n + C_4
proof (induction n rule: less_induct)
  case (less n)
  have 0 ≤ C_1 / (1 − a − c)
    using bounds by auto
  also have ... ≤ C_3
    by fact
  finally have C_3 ≥ 0 .

  show ?case
  proof (cases n > n_0)
    case False
    hence f n ≤ C_4
      using ineqs(4) by auto
    also have ... ≤ C_3 ∗ real n + C_4
      using bounds ‹C_3 ≥ 0› by auto
    finally show ?thesis .
  next
    case True
    have nonneg: a ∗ n ≥ 0 c ∗ n ≥ 0
      using bounds by simp_all
```

have (*max 0 b* + *1*) / (*1* − *a*) ≤ (*max 0 b* + *max 0 d* + *2*) / (*1* − *a* − *c*)
  using *bounds* **by** (*intro frac_le*) *auto*
hence n_0 ≥ (*max 0 b* + *1*) / (*1* − *a*)
  using *ineqs*(*1*) **by** *linarith*
hence *rec_less1*: *nat* ⌈*a*∗*n*+*b*⌉ < *n*
 using *bounds* ‹*n* > n_0› **by** (*intro akra_bazzi_light_aux1*[*of* _ n_0]) *auto*

have (*max 0 d* + *1*) / (*1* − *c*) ≤ (*max 0 b* + *max 0 d* + *2*) / (*1* − *a* − *c*)
  using *bounds* **by** (*intro frac_le*) *auto*
hence n_0 ≥ (*max 0 d* + *1*) / (*1* − *c*)
  using *ineqs*(*1*) **by** *linarith*
hence *rec_less2*: *nat* ⌈*c*∗*n*+*d*⌉ < *n*
 using *bounds* ‹*n* > n_0› **by** (*intro akra_bazzi_light_aux1*[*of* _ n_0]) *auto*

have *f n* = *f* (*nat* ⌈*a*∗*n*+*b*⌉) + *f* (*nat* ⌈*c*∗*n*+*d*⌉) + C_1 ∗ *n* + C_2
  using ‹*n* > n_0› **by** (*subst rec*) *auto*
also have ... ≤ (C_3 ∗ *nat* ⌈*a*∗*n*+*b*⌉ + C_4) + (C_3 ∗ *nat* ⌈*c*∗*n*+*d*⌉ + C_4) + C_1 ∗ *n* + C_2
  using *rec_less1 rec_less2* **by** (*intro add_mono less.IH*) *auto*
also have ... ≤ (C_3 ∗ (*a*∗*n*+*max 0 b*+*1*) + C_4) + (C_3 ∗ (*c*∗*n*+*max 0 d*+*1*) + C_4) + C_1 ∗ *n* + C_2
  using *bounds* ‹C_3 ≥ *0*› *nonneg* **by** (*intro add_mono mult_left_mono order.refl*; *linarith*)
also have ... = C_3 ∗ *n* + ((C_3 ∗ (*max 0 b* + *max 0 d* + *2*) + *2* ∗ C_4 + C_2) −
          (C_3 ∗ (*1* − *a* − *c*) − C_1) ∗ *n*)
  by (*simp add*: *algebra_simps*)
also have ... ≤ C_3 ∗ *n* + ((C_3 ∗ (*max 0 b* + *max 0 d* + *2*) + *2* ∗ C_4 + C_2) −
          (C_3 ∗ (*1* − *a* − *c*) − C_1) ∗ n_0)
  using ‹*n* > n_0› *ineqs*(*2*) *bounds*
 by (*intro add_mono diff_mono order.refl mult_left_mono*) (*auto simp*: *field_simps*)
also have (C_3 ∗ (*max 0 b* + *max 0 d* + *2*) + *2* ∗ C_4 + C_2) − (C_3 ∗ (*1* − *a* − *c*) − C_1) ∗ n_0 ≤ C_4
  using *ineqs bounds* **by** (*simp add*: *field_simps*)
finally show *f n* ≤ C_3 ∗ *real n* + C_4
  by (*simp add*: *mult_right_mono*)
 qed
qed

lemma *akra_bazzi_light*:

fixes f :: *nat* $\Rightarrow$ *real*
fixes n_0 :: *nat* **and** $a\ b\ c\ d\ C_1\ C_2$ :: *real*
assumes *bounds*: $a > 0$ $c > 0$ $a + c < 1$ $C_1 \geq 0$
assumes *rec*: $\forall n > n_0.\ f\ n = f\ (nat\ \lceil a{*}n{+}b\rceil) + f\ (nat\ \lceil c{*}n{+}d\rceil) + C_1 * n + C_2$
shows $\exists C_3\ C_4.\ \forall n.\ f\ n \leq C_3 * real\ n + C_4$
proof $-$
define n_0' **where** $n_0' = max\ n_0\ (nat\ \lceil(max\ 0\ b + max\ 0\ d + 2) / (1 - a - c) + 1\rceil)$
define C_4 **where** $C_4 = Max\ (f\ `\ \{..n_0'\})$
define C_3 **where** $C_3 = max\ (C_1 / (1 - a - c))$
$((C_1 * n_0' + C_2 + C_4) / ((1 - a - c) * n_0' - max\ 0\ b - max\ 0\ d - 2))$

have $f\ n \leq C_3 * n + C_4$ **for** n
proof (*rule akra_bazzi_light_aux2*[*OF bounds* _])
show $\forall n > n_0'.\ f\ n = f\ (nat\ \lceil a{*}n{+}b\rceil) + f\ (nat\ \lceil c{*}n{+}d\rceil) + C_1 * n + C_2$
using *rec* **by** (*auto simp*: n_0'_*def*)
next
show $C_3 \geq C_1 / (1 - a - c)$
and $C_3 \geq (C_1 * n_0' + C_2 + C_4) / ((1 - a - c) * n_0' - max\ 0\ b - max\ 0\ d - 2)$
by (*simp_all add*: C_3_*def*)
next
have $(max\ 0\ b + max\ 0\ d + 2) / (1 - a - c) < nat\ \lceil(max\ 0\ b + max\ 0\ d + 2) / (1 - a - c) + 1\rceil$
by *linarith*
also have $\ldots \leq n_0'$
by (*simp add*: n_0'_*def*)
finally show $(max\ 0\ b + max\ 0\ d + 2) / (1 - a - c) < real\ n_0'$ **.**
next
show $\forall n \leq n_0'.\ f\ n \leq C_4$
by (*auto simp*: C_4_*def*)
qed
thus *?thesis* **by** *blast*
qed

lemma *akra_bazzi_light_nat*:
fixes f :: *nat* $\Rightarrow$ *nat*
fixes n_0 :: *nat* **and** $a\ b\ c\ d$ :: *real* **and** $C_1\ C_2$ :: *nat*
assumes *bounds*: $a > 0$ $c > 0$ $a + c < 1$ $C_1 \geq 0$
assumes *rec*: $\forall n > n_0.\ f\ n = f\ (nat\ \lceil a{*}n{+}b\rceil) + f\ (nat\ \lceil c{*}n{+}d\rceil) + C_1 * n + C_2$

```
 shows $\exists\, C_3\ C_4.\ \forall\, n.\ f\ n \le C_3 * n + C_4$
proof –
 have $\exists\, C_3\ C_4.\ \forall\, n.\ real\ (f\ n) \le C_3 * real\ n + C_4$
  using assms by (intro akra_bazzi_light[of a c $C_1$ $n_0$ f b d $C_2$]) auto
 then obtain $C_3$ $C_4$ where le: $\forall\, n.\ real\ (f\ n) \le C_3 * real\ n + C_4$
  by blast
 have $f\ n \le nat\ \lceil C_3 \rceil * n + nat\ \lceil C_4 \rceil$ for n
 proof –
  have $real\ (f\ n) \le C_3 * real\ n + C_4$
   using le by blast
  also have $\ldots \le real\ (nat\ \lceil C_3 \rceil) * real\ n + real\ (nat\ \lceil C_4 \rceil)$
   by (intro add_mono mult_right_mono; linarith)
  also have $\ldots = real\ (nat\ \lceil C_3 \rceil * n + nat\ \lceil C_4 \rceil)$
   by simp
  finally show ?thesis by linarith
 qed
 thus ?thesis by blast
qed

lemma T'_mom_select_le': $\exists\, C_1\ C_2.\ \forall\, n.\ T'\_mom\_select\ n \le C_1 * n +$
$C_2$
proof (rule akra_bazzi_light_nat)
 show $\forall\, n>20.\ T'\_mom\_select\ n = T'\_mom\_select\ (nat\ \lceil 0.2 * n + 0 \rceil)$
$+$
            $T'\_mom\_select\ (nat\ \lceil 0.7 * n + 3 \rceil) + 17 * n + 50$
  using T'_mom_select.simps by auto
qed auto

end
```

50 Bibliographic Notes

Red-black trees The insert function follows Okasaki [15]. The delete function in theory *RBT_Set* follows Kahrs [11, 12], an alternative delete function is given in theory *RBT_Set2*.

2-3 trees Equational definitions were given by Hoffmann and O'Donnell [9] (only insertion) and Reade [19]. Our formalisation is based on the teaching material by Turbak [22] and the article by Hinze [8].

1-2 brother trees They were invented by Ottmann and Six [16, 17]. The functional version is due to Hinze [7].

AA trees They were invented by Arne Anderson [3]. Our formalisation follows Ragde [18] but fixes a number of mistakes.

Splay trees They were invented by Sleator and Tarjan [21]. Our formalisation follows Schoenmakers [20].

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.* [14].

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] R. Hinze. On constructing 2-3 trees. *J. Funct. Program.*, 28:e19, 2018.

[9] C. M. Hoffmann and M. J. O'Donnell. Programming with equations. *ACM Trans. Program. Lang. Syst.*, 4(1):83–112, 1982.

[10] 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.

[11] S. Kahrs. Red black trees. http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html.

[12] S. Kahrs. Red-black trees with types. *J. Functional Programming*, 11(4):425–432, 2001.

[13] T. Nipkow. Automatic functional correctness proofs for functional search trees. http://www.in.tum.de/~nipkow/pubs/trees.html, Feb. 2016.

[14] 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.

[15] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[16] T. Ottmann and H.-W. Six. Eine neue Klasse von ausgeglichenen Binärbäumen. *Angewandte Informatik*, 18(9):395–400, 1976.

[17] T. Ottmann and D. Wood. 1-2 brother trees or AVL trees revisited. *Comput. J.*, 23(3):248–255, 1980.

[18] 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.

[19] C. Reade. Balanced trees with removals: An exercise in rewriting and proof. *Sci. Comput. Program.*, 18(2):181–204, 1992.

[20] B. Schoenmakers. A systematic analysis of splaying. *Information Processing Letters*, 45:41–50, 1993.

[21] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[22] F. Turbak. CS230 Handouts — Spring 2007, 2007. http://cs.wellesley.edu/~cs230/spring07/handouts.html.